EXHIBIT 12

Privacy Analysis of Samsung's Crowd-Sourced Bluetooth Location Tracking System

Tingfeng Yu
School of Computing
The Australian National University
Canberra, ACT, Australia

James Henderson
School of Computing
The Australian National University
Canberra, ACT, Australia

Alwen Tiu
School of Computing
The Australian National University
Canberra, ACT, Australia

Thomas Haines
School of Computing
The Australian National University
Canberra, ACT, Australia

ABSTRACT

We present a detailed privacy analysis of Samsung's Offline Finding (OF) protocol, which is part of Samsung's Find My Mobile (FMM) location tracking system for locating Samsung mobile devices, such as Samsung smartphones and Bluetooth trackers (Galaxy SmartTags). The OF protocol uses Bluetooth Low Energy (BLE) to broadcast a unique beacon for a lost device. This beacon is then picked up by nearby Samsung phones or tablets (the *finder* devices), which then forward the unique beacon, along with the location it was detected at, to a Samsung managed server. The owner of a lost device can then query the server to locate their device. We examine several security and privacy related properties of the OF protocol and its implementation, from the perspectives of the owner, the finder and the vendor. These include examining: the possibility of identifying the owner of a device through the Bluetooth data obtained from the device, the possibility for a malicious actor to perform unwanted tracking against a person by exploiting the OF network, the possibility for the vendor to de-anonymise location reports to determine the locations of the owners or the finders of lost devices, and the possibility for an attacker to compromise the integrity of the location reports. Our findings suggest that there are privacy risks on all accounts, arising from issues in the design and the implementation of the OF protocol.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**; *Security protocols.*

KEYWORDS

Location Privacy, Mobile Devices, tracking tags, Bluetooth, SmartTags

1 INTRODUCTION

Portable devices such as smart phones and tablets often come with a feature that allows their owner to find those devices when they are lost, typically through the use of a web portal provided by their vendors. For such a feature to work, the owner would have to grant a permission for the device to share their locations periodically with the vendor. Examples of such a lost-device finding feature include Google's Find My Device,[1] Samsung's Find My Mobile (FMM) [2] and Apple's Find My.[3] A typical requirement for such a feature to work is that the lost device must be connected to the internet so that it can send its location report to a vendor server in the event that its owner flags the device as lost. It often comes with additional features such as playing sound on the device, locking the device or wiping off its data remotely.

In recent years, mobile device manufacturers such as Samsung and Apple have extended their lost-device tracking systems with an *offline finding* (OF) feature, which allows a lost mobile device to be found even when it does not have an internet connection. Both Apple and Samsung OF features share two key features: the use of Bluetooth Low Energy (BLE) for short range transmission of data between devices of a vendor, and crucially, an extensive network of (internet-connected) mobile devices (which we call *finder devices*) that relay location information to a vendor controlled server. We refer to the latter as the *OF network*. The basic idea is quite simple: when a (lost) device loses its internet connection, it starts broadcasting a unique beacon over BLE, which is then picked up by nearby finder devices participating in the OF network, who then forward the beacon and the location it is found to a vendor server. In this work, we are mainly concerned with Samsung's FMM Offline Finding (OF) feature,[4] which was introduced in 2020. An owner can track their devices' locations through Samsung's proprietary "Find My Mobile" (FMM) application running in a Samsung mobile device (e.g., a phone or a tablet), provided that the FMM feature is enabled for the device.

In 2021, Samsung released the Galaxy SmartTag,[5] which is a small BLE tracker that can be attached to various items, such as bags, keys, etc., to keep track of their locations and to find them when lost. Unlike smart phones or tablets, SmarTags are designed exclusively to be used as a tracking device, with no internet connectivity. So they rely crucially on the OF network to allow for long range location tracking (outside the range of BLE). SmartTags are registered and controlled through SmartThings, which is an umbrella control and management platform for a large variety of smart devices and home appliances. OF is also supported for SmartTags

[1] https://www.google.com/android/find/
[2] https://findmymobile.samsung.com/
[3] https://support.apple.com/find-my
[4] https://www.samsung.com/au/apps/find-my-mobile/
[5] https://en.wikipedia.org/wiki/Samsung_Galaxy_SmartTag

Technical Report, 2023, ANU
2023. https://doi.org/XXXXXXX.XXXXXXX

using the "SmartThings Find" add-on which works in conjunction with FMM.

At a basic level, devices in Samsung's OF network can be categorized into three roles: the owner device, the finder device, and the lost device. A mobile device can be registered to the Samsung OF network through the FMM app, while a SmartTag can be registered through Samsung SmartThings app. Each registered device is linked to the owner account under which it was registered from. When a registered device loses internet connectivity, or in the case of SmartTags, when it is out of the BLE range from its owner device, it broadcasts certain data over BLE periodically. This data contains a rotating identifier, called the *privacy ID*, which is unique to the lost device and which, in theory, can only be linked to its owner by Samsung and the owner device. The finder devices consist of both Samsung devices (phones and tablets), and some third parties' devices with FMM enabled. An active finder device periodically scans for BLE advertisements from nearby FMM devices and reports the locations of those devices to a Samsung's server. The location reports of the lost devices will be downloaded onto the owner device when the owner queries the locations of their lost devices. The effectiveness of the OF feature depends on the size of its OF network, i.e., the number of phones or tablets participating in the network. In the case of Samsung OF network, it is estimated to have around 200 million active finder devices [12] in 2022.

This paper describes in detail the design and the implementation of Samsung's OF protocol for FMM and SmartTags, and analyse its security and privacy. We focus on the following research questions:

(RQ1) *Identification of an FMM device or a SmartTag.* Can an FMM device or a SmartTag be identified through its BLE data?

(RQ2) *Unwanted tracking.* Can Samsung OF network be misused for unwanted tracking of a person or an object by a party other than Samsung?

(RQ3) *End-to-end location privacy.* To what extent does the design of the OF network protocol protect the location privacy of the lost devices and the finder devices from the service provider (Samsung)?

(RQ4) *Location report integrity.* Is it possible for an actor (other than the owner and the vendor) to forge a location report of a lost device?

RQ1 centers around the privacy protection of the owner of an FMM device or a SmartTag against (long term or short term) tracking of their location through the BLE data emitted by the device, by a third party adversary (other than the owner and the vendor). RQ2 addresses a recent phenomenon, mostly associated with Apple AirTags, where the tags were used by their owner to stalk a person against their consent [8]. In this case the victim of an unwanted tracking may even be someone who does not own any devices from the vendor and thus not participating in the OF network. RQ3 raises the question as to what extent Samsung is aware of the movement of both the owners of the devices being tracked and the finder devices. Whereas Apple advertised their OF network as providing end-to-end privacy, in the sense that the service provider (Apple) has no way of recovering the location information sent through its OF network [1], Samsung provided no such claim as far as we are aware of. RQ4 is more of a security (integrity) issue rather than a privacy one. However, the possibility of disrupting location reports, under the right circumstances, can act as an ad hoc measure for addressing unwanted tracking (RQ2).

Summary of contributions. To the best of our knowledge, we are the first to provide a detailed analysis on Samsung offline-finding protocols and its privacy issues. More precisely, our contributions are the following:

- We provide a comprehensive reverse-engineering of Samsung OF protocols for both mobile devices and SmartTags. This includes the registration protocols for SmartTags, the cryptographic methods for generating unique identifiers for lost devices, the protocol for the finder devices to report location information to Samsung, and the protocol for the owners to query the location of their lost devices. This effort allows us to answer definitively the research questions raised above (RQ1 to RQ3).
- We identified several vulnerabilities in both FMM and SmartTags that would allow an attacker to link BLE packets observed from a target device over multiple observations, through BLE interactions only. This allows us to conclude definitively that long term identification of a device or a tag (RQ1) is possible through its BLE data only.
- Through our analysis of the OF protocols, we managed to impersonate completely a SmartTag to the OF networks. This opens the possibility of creating a custom tracking device that can be tuned to circumvent potential anti-tracking mechanisms by the vendor.
- Our analysis also confirmed that unwanted tracking (RQ2) is possible. This is, however, a rather easy observation as Samsung (at the time of writing) only implements a very basic anti-tracking mechanism that is targetted at observing BLE data from SmartTags. Given our result above, it is quite straightforward to circumvent this detection by crafting a custom tracking device leveraging Samsung's OF network.
- Our analysis of the registration process and the location report/querying protocols suggests that the vendor does indeed possess the information needed to link an account to a location report, so currently Samsung OF network does not guarantee end-to-end privacy for their users (RQ3). Moreover, the vendor server does not appear to check the integrity of the location reports, opening it to manipulation by third parties to forge the location reports (RQ4).

Coordinated disclosure

We have reported our findings related to RQ1, RQ2 and RQ4 to Samsung in late January 2022. One of the issues we raised concerns the small pool of privacy IDs being used for FMM BLE packets, which has now been publicly acknowledged and assigned SVE-2022-0126 ("Improper identifier creation logic in Find My Mobile ") and registered as CVE-2022-33707 at MITRE. Samsung claimed that they have fixed this issue in the July 2022 update to the FMM app, however at the time of writing, we have not yet tested whether the fix addresses the issues related to RQ1. Samsung has also issued us a bug bounty reward for this report. The issues affecting SmartTags have not yet been completely resolved. Samsung claimed they have addressed some of the issues we raised in a firmware update to SmartTags, but we have not yet performed detailed analysis on the

updates. Samsung did confirm that they would not fix the issue of de-anonymisation through BLE DFU mode (see Section 5.1.3) as they claimed it would interfere with their device firmware download and update process at their repair centre. We have allowed ample time for Samsung to address these issues, exceeding the industry standard of 90-day embargo period, hence the publication of these details.

Related work

We now provide a literature review on existing security and privacy analysis of Samsung and Apple's Bluetooth trackers, Offline Finding (OF) networks, continuity protocols, and other relevant products that implement Bluetooth technology.

Apple's Offline Finding Network. The closest to our work is the security and privacy analysis of Apple's FindMy offline-finding network by Heinrich, et. al. [15]. Their study uncovered two design and implementation flaws outside Apple's adversary model that could lead to location correlation attacks and unauthorized access to location histories of the past week. They reverse-engineered FindMy protocols and showed that one could create custom tracking devices leveraging on the FindMy network through their OpenHaystack framework.[6]

Samsung FMM App. Researchers at Char49 discovered several vulnerabilities [7] in an earlier version of Samsung FMM app, allowing, among others, a malicious app installed in the phone to manipulate the URL endpoint accessed by the FMM app, and to access unprotected broadcast receivers in the FMM app. This analysis was done prior to the introduction of the offline-finding features to FMM, so it did not cover the OF related vulnerabilities.

Hardware and firmware security of AirTags and SmartTags. Both Apple AirTags and Samsung SmartTags are implemented using the nRF52 series of System on Chips (SoC). The nRF52 series have been used for a wide range of Internet of Things (IoT) devices; they support a variety of wireless communication protocols, such as Bluetooth LE and Bluetooth Mesh. However, the nRF52 series chips are known to be vulnerable to power glitching attacks. AirTags use the nRF52832 chip for BLE and Near Field Communication (NFC) connectivity. Roth et. al. analysed the hardware and the firmware security of AirTags, and documented AirTags' communication protocols in detail [19]. The main firmware of the AirTag was extracted through voltage glitching attacks on its nRF chip. By reprogramming the firmware and changing the configuration data, they were able to

- modify the internal behavior of AirTags, including cloning an AirTag, customizing the soundset of the AirTag, using the AirTag's accelerometer as microphone;
- change the BLE and NFC behavior of AirTags which can potentially be exploited for malicious purposes.

By instrumenting the iPhone-AirTag interface, they were also able to unlock undocumented commands and features on AirTags over-the-air without hardware modification.

Galaxy SmartTags use the Nordic nRF52833 chip. Luca Bongiorni exploited a voltage fault injection vulnerability on the nRF52833 chips to dump the firmware of SmartTags and released the dumped firmware images and information related to the attack,[7] although as far as we know the author did not attempt a reverse engineering of the OF protocol for SmartTags.

Bluetooth trackers from other vendors. Apple and Samsung are relatively newcomers when it comes to bluetooth tracking dvices. There were already a number of bluetooth trackers in the market prior to the introduction of AirTags and SmartTags, notably the Tile tracker; see Weller et. al. [21] for a recent survey on these trackers. Weller et. al. also presented a detailed analysis of the security and privacy aspects of various commercial Bluetooth trackers, including Nut, Smart Tracker, Tile, Musegear finder, iTrackEasy, Cube Tracker, Keeper, iTracing, iSearching, and FindELFI, focusing on the interactions of these finders, their associated mobile apps and the backend cloud servers for crowdsourced location tracking. However, they did not analyse the privacy issues arising from the BLE protocols used in these trackers.

Anti-Tracking Technologies. Apple's FindMy network consists of hundreds of millions of active devices, which has raised a concern on whether an attacker can abuse the network for malicious tracking. Apple has developed and implemented an in-built anti-tracking framework, which would send users a safety alert if it is detected that they have been followed by an unknown FindMy tracker.

In 2021, Mayberry et al. analysed the effectiveness of Apple's in-built anti-tracking mechanisms, then developed and confirmed three techniques to defeat the mechanisms [17]. The first technique is Bit Flipping. The OF advertisement data of FindMy supported devices follows a fixed structure, where type of the device is stored in byte 2 of the advertisement data. Mayberry et al. found that when byte 2 is set to 0x00, which indicates that the device type is iPhone, FindMy would not report the device as a tracker regardless of its tracking period and distance. A legitimate FindMy device broadcasts OF data containing a rolling key shared between the owner and the device when it is away from the owner and performs MAC address randomization and advertisement data rotation in-sync every 24 hours. The other two techniques are both based on frequent Key Rotations to prevent anti-tracking algorithms from identifying a tracker device based on the key. The difference is that in the second technique, a new key is selected from a large pre-generated set of valid keys when rotating the advertisement data. In the last technique, each new key is generated deterministically using the rolling key generation algorithm used by FindMy devices. Mayberry et al.'s study has shown that the iOS tracking detection is unable to detect FindMy trackers with fast advertisement payload rotations or mark devices broadcasting OF data in the lost iPhone format as trackers. Therefore, an adversary can easily bypass Apple's anti-tracking mechanism by customizing a Bluetooth capable device that implements either of the above techniques and track a target without being detected.

AirGuard is an anti-tracking application designed and developed by researchers from SEEMOO lab [14]. AirGuard is an open-sourced Android application that was mainly designed to protect Android users from BLE trackers that leverage on Apple's OF network. The experiment results show that AirGuard achieved a higher success

[6] https://github.com/seemoo-lab/openhaystack

[7] https://github.com/whid-injector/Samsung-SmartTag-Hack

rate, a lower false positive rate, and lower notification delay in identifying and reporting trackers under various tracking scenarios in comparison to Apple's in-built anti-tracking feature. The AirGuard categorizes devices into 5 types: Apple device, Airpod, FindMy accessory, AirTag, and Tile tracker. At the time of writing, AirGuard does not yet support detection of SmartTags.

Analysis of Apple's Continuity Protocols. Continuity protocols are underlying protocols used for Apple's continuity services, which aim to allow different devices within Apple's ecosystem to share data seamlessly. The protocols rely on BLE for data exchange. Celosia et. al. [6] reverse engineered Apple's continuity protocols and uncovered several implementation issues that can be used for passive and active tracking attacks, including: broadcast of Bluetooth data on identity addresses, infrequent randomization of MAC addresses, identifiers and other sensitive information contained in BLE advertisement data. Martin et. al. [16] discovered that the BLE advertisement of HandOff messages in the continuity protocols contains a predictable incremental sequence.

These found issues would result in privacy leaks with severity ranging from low (e.g., leakage of battery level) to high (e.g., leakage of phone number in plaintext) and defeats the anti-tracking provisions in BLE as the leaked data can be used by an adversary to perform long-term tracking of a device passively.

General BLE Related Security and Privacy Issues. A Bluetooth device can be uniquely identified by its Bluetooth MAC address. Therefore, to avoid long-term tracking of a Bluetooth device, vendors often implement anti-tracking mechanisms, such as randomization of a device's Bluetooth MAC address periodically.

However, multiple privacy issues have been found in the current implementations of BLE advertising mechanism and GATT profiles content of Bluetooth devices, which can be utilized to defeat such anti-tracking mechanisms or to retrieve sensitive information that could affect the owner of the device.

Celosia et. al. analysed the data exposed in the GATT profiles based on a large dataset of GATT profiles collected in daily environments [4]. It was shown that content of a GATT profile may contain identifiers or diverse data that act as a fingerprint of the device, which allows long-term tracking of a Bluetooth device regardless of the MAC address randomization. The result has also shown that the data exposed within a GATT profile may contain sensitive information to be inferred, such as health data, which would violate the privacy of the device owner.

Celosia et. al.'s privacy analysis on the current implementations of BLE advertising mechanism has shown multiple common implementation mistakes that could result in privacy threats to the device or the device's owner [5]. (1) Many devices still use a stable Bluetooth MAC address, which is vulnerable to long-term tracking. (2) Devices that implements MAC address randomization may contain unique data in the BLE advertisement packets that allows the device to be identified and tracked. (3) The interval of the MAC address randomization exceeds the recommended maximum duration of 15 minutes.

Outline

The remainder of the paper is structured as follows: We give a brief overview of some basic cryptographic operations related to Samsung's OF protocols in Section 2. In Section 3 and 4, we present the technical details of OF protocols for FMM and SmartTags that result from our reverse engineering efforts. In Section 5, we perform a security and privacy analysis on Samsung's OF feature based on our findings discussed in previous sections and list each vulnerabilities. Finally, Sections 6 and 7 concludes our work with a discussion of the impact.

2 BACKGROUND

This section gives a very brief overview of the relevant cryptographic functions used in the Samsung OF protocols and some basic concepts related to BLE.

2.1 ECDH key exchange and AES block cipher

There are two main cryptographic constructions used in the OF protocol: the Elliptic-curve Diffie-Hellman (ECDH) key exchange protocol and the AES block cipher and its associated encryption modes. The ECDH protocol is a key exchange protocol that allows two participants in a protocol, without a prior shared secret, to derive a common secret, that can be used to derive other keys, e.g., for encryption or for message authentication. The AES block cipher is a symmetric encryption algorithm that is widely used for data encryption, and as a building block for other cryptographic functions. We explain briefly each of these constructions. For further details, we refer the interested reader to [13] (for ECDH) and [9] for the AES algorithm.

The ECDH builds on the Diffie-Hellman key exchange protocol [10], where the underlying group operations are defined over an elliptic curve. For our purposes, an elliptic curve (EC) is a plane curve over a finite field F_q. It can be defined by the set of points $(x, y) \in F_q \times F_q$ that satisfies the Weierstrass equation $E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, if certain conditions hold [13]. Given an EC point P and a scalar k, scalar multiplication $kP = Q$ can be computed in a few steps using a combination of scalar multiplication and point addition. In contrast, computing the discrete logarithm $\log_P(Q)$, such as the reverse operation: finding $k \in Z$ such that $Q = kP$ is hard and considered infeasible when a sufficiently large finite field F_q and a curve with carefully selected domain parameters were used. This asymmetry in the computational complexity encodes the security assumption for elliptic curve cryptography (ECC), where the security level is based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP).

In ECC, a private key k is an integer. The corresponding public key P is computed by $P = kG$, where G is the generator point, a special pre-defined EC point that any point in its EC subgroup can be generated by scalar multiplication of G; p is the order of the curve, which defines the finite field F_q the curve is over.

ECC is often combined with the traditional Diffie-Hellman protocol for establishing shared keys over public channels. A simple key exchange procedure using Elliptic-curve Diffie–Hellman (ECDH) can represented as follows:

- The generator point G and the order p are public parameters.

- Peer A has a private key a and the corresponding public key $A = aG$. Peer B has a private key b and the corresponding public key $B = bG$.
- Public key exchange: Peer A and B exchange their public keys over an authenticated public channel.
- Shared key computation: Peer A then uses the public key B from peer B and the private key a to arrive the secret key $A_{key} = aB = abG$. Peer B performs the same computation process using private key b and peer A's public key A, which would produce the same key $B_{key} = bA = abG = A_{key}$.

Samsung's OF implementation of ECDH uses the elliptic curve Curve25519 [2], which was designed to achieve high speeds at computation without compromising the security strength. It is defined by the Montgomery equation $y^2 = x^3 + 486662x^2 + x$ over the prime field of order $2^{255} - 19$ with a generator point $G = 9$.

The Advanced Encryption Standard (AES) algorithm [9] is a symmetric cipher, which means that the same key is used for the encryption and decryption process. Hence, the same cryptographic key must be shared between the sender and the recipient in order for them to communicate securely. AES offers cryptographic keys with size 128, 192 and 256 bits. The cipher encrypts/decrypts plaintext/ciphertext in blocks of 128 bits data.

AES, like all block-ciphers, has multiple standardized mode of operations, where each mode describes a way to apply the single-block operation of the cipher to securely transform data longer than the block size. To encrypt data that is not in blocks of 128 bits using AES, the padding process must be applied to the last block to extend the data to a multiple of the block size. Samsung's FMM and SmartTags implements AES CBC mode cipher with PKCS#7 padding scheme [18] to encrypt/decrypt data for various OF related operations. If the last block of a plaintext has a length of $16-n$ bytes, the PKCS#7 standard specifies that n bytes need to be appended to the plaintext, where each padded byte has a value of n (in hex).

The CBC encryption mode uses a 128-bit initialisation vector (IV), that is often used to introduce some nondeterminism in the ciphertext. Given a plaintext with n blocks $(P_0, \ldots, P_{n-1})$, an IV and an encryption key k, the CBC mode produces the ciphertext blocks $C_0, \ldots, C_{n-1}$ as follows:

$$C_0 = E_k(P_0 \oplus IV)$$
$$C_i = E_k(P_i \oplus C_{i-1}), \text{ for } i \in [1, \ldots, n)$$

where E_k refers to the AES blockcipher algorithm with key k and $\oplus$ refers to the bitwise exclusive-OR (XOR) operation. The IV is XORed with the first block of plaintext (P_0) to introduce some non-determinism. Then it is encrypted to create the first ciphertext block (C_0). Each subsequent plaintext block (P_i) is XORed with the previous ciphertext block (C_{i-1}) then encrypted using the encryption key.

The AES algorithm is also used in the Bluetooth Low Energy (BLE) Specification for key generation, which will be discussed in Section 2.2.4.

AES-CBC can also be used as a Message Authentication Code (MAC) in which case the IV should be fixed, or at least unpredictable and uncontrolled by the adversary, and care must be taken to prevent message extension attacks. The use of AES-CBC in OF is very strange as well shall elaborate on later. It is configured in such a way that it is neither used in a standard manner for encryption nor authentication and hence provides dubious security for both purposes. Given that underlying hardware did support authenticated encryption modes, we are mystified as to why Samsung decided to use AES-CBC in such a non-standard way.

2.2 Bluetooth Low Energy (BLE)

SmartTags uses Bluetooth Low Energy (BLE) [11] for data transmission. BLE is a wireless communication technology designed for short-range data transmission. It has been widely applied to small battery-powered devices that do not require continuous streaming of data, such as fitness trackers and smartwatches.

2.2.1 BLE protocol stack. The protocol stack of BLE can be broken down into three blocks: application, host, and controller. The Application is the highest layer in the protocol stack, it is responsible for the data handling, application logic, and human-machine interface. The architecture is dependent on the specific use-case and implementation. The Host is a software stack that consists of the upper layers and profiles in the BLE protocol stack. Each profile defines ways for certain protocols in the stack to interact with each other or work together. The Controller is a subsystem that consists of the lower layers in the protocol stack. It is responsible for generating/receiving, modulation and demodulation of RF signals. This section will provide details on the two profiles contained in the Host block, which are the Generic Attribute Profile (GATT) and the Generic Access Profile (GAP).

GAP provides guidelines for the advertising and connection functionalities that any BLE implementations must follow. A BLE device needs to operate in one or more roles to participate in the BLE network, and it may operate in multiple roles simultaneously. These roles are:

- Advertiser: a device that sends out BLE data that is available to any nearby Bluetooth capable devices.
- Observer: a device that listens to BLE advertisement data and may process the data from Advertisers. An observer is not connected to any advertisers.
- Central: A central is a device that initiates a connection after first receiving advertisement data from an advertising peripheral. A central can connect to multiple peripheral devices.
- Peripheral: A peripheral is a device that advertises BLE data to announce its presence to the centrals and accepts the incoming connection from a central. Upon connection, the peripheral device stops broadcasting BLE data and remains undiscoverable throughout the connection.

GATT defines how data is organized and exchanged over an established connection between BLE devices. It is build on top of Attribute Profile (ATT) protocol that provides the mechanisms for data exchange in the form of "attributes". An attribute is defined by a 4-byte handle, a 128-bit UUID, a set of permissions, and a value. The attribute handle is used as the identifier to access the attribute value. The UUID specifies the type of data the attribute contains.

GATT extends ATT by defining different types of attributes to provide a hierarchical structure for organising user data, which include services, characteristics, and descriptors, as detailed in Figure 1.

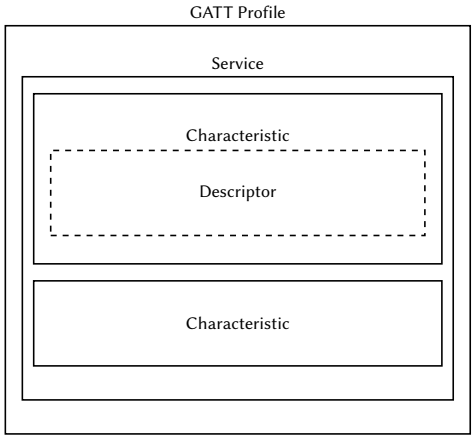

Figure 1: Content of the GATT profile

Each service groups conceptually related characteristics together, and each characteristic is a container of user data. A characteristic can be followed by descriptors, which provide additional metadata of the characteristic and its value.

2.2.2 BLE Communication. BLE has two ways of transferring data: advertising over BLE and data exchange over connections.

Advertising is the process of a BLE device sending out data packets in one-way. BLE supports several advertisement types. The advertisement type SmartTags use is ADV_IND (connectable, undirected advertising). Undirected means that the data is broadcasted and is accessible by any nearby Bluetooth-capable device instead of being targeted at a specific Bluetooth address. Connectable means that the advertiser would accept connection requests.

Communication over connections allows bidirectional data transfer between the peripheral and the central. Data packets are exchanged through characteristics in the GATT server of the peripheral device.

2.2.3 Bluetooth MAC address. A Bluetooth MAC address is a 48-bit value that can be used to identify a device. There are four types of MAC addresses:

- Public Address
- Random Static Address
- Random Private Non-Resolvable Address
- Random Private Resolvable Address (RPA)

A Public Address never changes and is registered with IEEE to ensure that each public address is unique worldwide. A Random Static Address does not require registration with IEEE; it remains static during the runtime of a device. Each Bluetooth-capable device has an *Identity Address*, which is either a Public Address or Random Static Address.

The two types of Random Private Addresses (Non-Resolvable, Resolvable) are primarily used for privacy protection. The identity address of a BLE device can be hidden by using the Random Private Address (RPA), which prevents long-term tracking of the device. Random Private Non-Resolvable Addresses aim to prevent a device from being identified by any other devices; Random Private Resolvable Addresses (RPAs), on the other hand, only prevent a device from being identified by non-trusted parties, while trusted parties can still identify the device using a shared key: the Identity Resolving Key (IRK), which will be elaborated in §2.2.4.

Algorithm 1 The Hash Function

function AH(IRK,$prand$)
  $r' \leftarrow 0x000000 \,||\, prand$
  $hash \leftarrow e(IRK, r') \bmod 2^{24}$
  return $hash$
end function

2.2.4 BLE pairing and address resolution. Pairing is the process by which two BLE devices exchange necessary information so that an encrypted connection can be established. BLE has several pairing modes, which are determined by the authentication requirements and input/output (IO) capabilities of the devices participating in the pairing process. As part of pairing, two keys are exchanged:

- Identity Resolving Key (IRK): to generate/resolve Random Private Resolvable Addresses (RPA)
- Long Term Key (LTK): to encrypt/decrypt the connection session

The IRK is used to generate and to resolve an RPA. The three most significant bytes of an RPA correspond to a random number `prand`. The three least significant bytes correspond to the hash value of `prand`.

As defined in the Bluetooth Core Specification [11], the hash function function `ah` (shown in Algorithm 1) is used to compute the hash value using the IRK and `prand` as input. It first extends the 3-byte `prand` to a 16-byte array `r'` so that the length matches the block size of AES. Then, it calls the security function `e` which encrypts `r'` with an AES/ECB cipher using the IRK as the key then returns the ciphertext. The hash value is obtained by modding the ciphertext with 2^{24}.

3 OFFLINE FINDING PROTOCOL FOR SMARTPHONES

In this section, we discuss our findings in the reverse engineering of offline-finding features of Samsung Find My Mobile (FMM) app. The results of this section apply to all versions of FMM app (with the offline finding features) prior to version 7.2.24.12 (July 2022).

3.1 Methodology

Four main investigative methods have been used: Android APK Reverse Engineering, passive BLE scanning, active BLE interaction and device logs. These methods have been used both in isolation and in conjunction to better inform the other methods and gain a clearer overall understanding of the protocol and its operations. Initial investigations primarily consisted of passive BLE scanning and source code review, with later investigations adding device logs and active BLE interaction methods.

3.1.1 Android APK Reverse Engineering. The JADX decompiling tool[8] was used to decompile the FMM Android application package (apk) into its Java source code. The reverse engineering process

[8] https://github.com/skylot/jadx

allowed the operation of the FMM application relating to Offline Finding to be understood and analysed in details.

3.1.2 BLE Scanning. The passive BLE scanning involved observation and analysis of BLE packets obtained from offline device's BLE advertisements. Two main scanning methods were used which offer slightly different functionalities: (1) Device's in-built Bluetooth Hardware – allows capture of Bluetooth advertising data; (2) Ubertooth One Bluetooth Sniffing Device [9] – allows capture of Bluetooth advertisements as well as sniff communications on already established (or newly establishing) Bluetooth connections.

3.1.3 Active BLE Interaction. The Bluetooth GATT profile setup by offline devices can be interacted with using a BLE capable device through reading or writing to the characteristics offered. Interacting with these characteristics and observing the reaction of the phone allowed us to gain more insight into the protocol.

3.1.4 Device Logs. We analysed the run-time output of the device (main system logs) and the FMM application (application logs) using `adb logcat`. These logs are viewed to better understand the workings of the application when it is interacted with, either through active BLE interaction or interacting with the screen itself.

3.2 Find My Mobile in Details

The OF protocol has multiple modes of operations that depend on the functions supported by the devices involved as well as the type of device to be located. In this section, we outline the main OF protocol that applies to mobile devices, which consists of four main operations: Device/Account Registration, Offline (Lost) Device Operation, Online (Helper) Device Operation, and Device/Account Deregistration.

The protocol can be summarised as follows. Initially, devices must complete the registration process with an active network connection. All devices involved with the protocol must be registered to OF. When a registered device goes offline, it starts advertising a unique payload that identifies itself. This payload is picked up by nearby online (registered) devices which parse the payload extracting the device's identifier. The online device then accesses available location services to find out its own location. It then sends the lost device's identifier and the location through to Samsung. The owner of the lost device can then access the FMM web service to find out its location. Further details of each operation are outlined below.

Device/Account Registration. When a user decides to enable FMM, they are required to login to a Samsung account. Their device communicates with Samsung to verify the account and complete the sign in. This associates the device to the given account under the normal FMM operation.

If the user then decides to enable Offline Finding, a separate registration process is started. This registers the device and account with the Offline Finding feature by sending a HTTPS request to a specific 'registerDevice' URL within Samsung's servers. This process requires an active network connection so that the device can communicate with Samsung. The device will not let a user toggle OF on without an active network connection. The registration process is a single HTTPS request that receives a single HTTPS response.

To begin the registration process, the device firstly generates a 16-byte random secret key. This is generated using Java Random (notably not Java SecureRandom). This random secret key is not stored on the device. The device then forms a registration object which is to be sent to Samsung using HTTPS. This object contains the following device and account information:

- Secret key: 16-byte random generated using Java Random.
- Device ID: Base64 encoding of an MD5 hash of the device's IMEI. The IMEI is constant and unique for a device.
- User ID: A value associated with the Samsung account logged in.
- Device type: The type of device being registered. (either Tablet or Mobile).
- Region: The ISO country code associated with the device.
- Client version: The version of the FMM application running on the device.
- SDK version: The Android SDK version the device is running.
- Model name: The device model.

This information is then sent to Samsung via a HTTPS request. If the request is successful, then Samsung responds with a request result containing the following information:

- Policy: This contains policy settings for how the device should operate when using OF. It contains various intervals and windows for specific OF operations including advertising, scanning, payload shuffling. It also contains a maximum report count which indicates how many lost devices a helper device can report at one time.
- Target URLs: This contains six Samsung server URLs that are used for various FMM operations. Most importantly it includes the URL for reporting locations.
- `PrivateIDConfig`: This contains a secret key, an IV and a privacy ID pool size. This is the basis for the advertisements that an offline lost device generates.

The device then stores the information from this response to guide the OF operation. The `PrivateIDConfig` is the most important part for the OF protocol and will be discussed further. Receiving a successful response from Samsung marks the end of the registration process.

3.2.1 PrivateIDConfig. The `PrivateIDConfig` contains a secret key, an IV and a privacy ID pool size. It is unclear how the secret key response from Samsung relates to the original random 16-byte secret key that the device generates. The original secret key is not stored by the device leading to the assumption that it is not important and most likely just acts as a seed for the secret key that Samsung generates. The secret key is used as the base key in the generation of advertising data used when the device is offline.

The IV response from Samsung is a standard IV that is used for all devices. Throughout the investigation, the same IV was repeatedly seen in these responses. This IV being (in base64 encoding): `+IABCfvBZHJYFUek8vp3Gg==`. The implications of this are discussed further in the analysis section.

The privacy ID pool size determines the amount of possible advertising values the device will generate. This has also been

[9] https://greatscottgadgets.com/ubertoothone/

Table 1: Characteristics under the Authentication Service

Name	UUID
NONCE	A12BE31C-5B38-4773-9B9D-3D5735233A7C
ENONCE	4EBE81F6-B952-465E-9ECE-5CA39D4E8955
SUPPORTED_CIPHER	50F98BFD-158C-4EFA-ADD4-0A70C2F5DF5D

observed to be standard for all devices, taking the value 51. This means that there are only 51 possible advertisement payloads from a lost device and the implications of this are also discussed further in the analysis section.

3.2.2 Offline (Lost) Device Operation. When a OF registered device no longer has an active network connection, it enters 'Lost Mode' and triggers the Offline Finding service to start. The lost device then creates a GATT server profile and starts advertising on the main OF service UUID (FD69). The advertising is the fundamental operation for lost devices as part of the main OF protocol. The GATT server is not directly used by the main OF protocol, however it can be interacted with via BLE and is used as part of secondary OF protocols. This opens another attack surface, and so the profile is described below.

GATT Server Profile. The lost mode device creates a GATT server containing two primary services, the authentication service and the FME serive.

Authentication Service Service UUID `EEDD5E73-6AA8-46-73-8219-398A489DA87C` is used to implement a challenge-response protocol for a connected device to authenticate to the GATT server. It contains three characteristics (see Table 1):
The `SUPPORTED_CIPHER` characteristic is readable and contains information about the cipher to be used, which is AES/CBC/PKCS7. The `NONCE` characteristic is readable and returns an IV to be used during encryption. This IV is a random nonce that is generated using Java `SecureRandom` each time a client connects to the server. The `ENONCE` characteristic is writeable and expects to receive an encrypted version of the string "smartthings". This string must be encrypted using the given IV and with the device's secret key (from the `PrivateIDConfig`). Writing the correct ciphertext to the encrypted nonce characteristic completes the handshake between client and server.

FME Service After completing the handshake, the client is now authenticated and can interact with the characteristics in the FME service. The device's alarm can be set to ring by writing the byte 01 (encrypted using the same cipher) to the `ALARM` characteristic (UUID 4a1351bb-d617-4612-a8e3-8dee6ca13e7b).

Lost Mode Advertising. the lost mode advertisements are the fundamental part of the OF protocol. The lost device generates an advertisement containing a unique identifying payload (the private ID) which is picked up by a helper and reported to Samsung. The private ID is generated from the device's `PrivateIDConfig` as follows:

The cipher is initialised with an encryption key and an IV. The encryption key is the device's secret key from the `PrivateIDConfig`, and the IV used is the standard IV `+IABCfvBZHJYFUek8vp3Gg==`. The cipher encrypts a 20-byte array consisting of the device's secret key and an extra four bytes that are based off a random nonce (i). These four bytes provide all of the non-determinism to the private ID generation. The random nonce is simply an integer generated by Java Random that is bounded to the privacy ID pool size (51). i.e. the random nonce is always a value between 1 and 51.

The four random bytes are actually only two unique bytes which are appended to the start and the end of the secret key. The first of the unique bytes is generated by a bitwise right shift of i by 8-bits, followed by a bitwise AND with 255. Since i is between 1 and 51, this always results in 0. The second unique byte is generated by a bitwise AND between i and 255, which results in i. Table 2 shows the structure of the 20-byte array that is fed into the cipher.

Table 2: Byte Array Encrypted During Lost Mode Advertising Data Generation

Byte 0	Byte 1	Bytes 2-17	Byte 18	Byte 19
00	i	secret key	00	i

The AES/CBC/PKCS7 encryption is performed, and the first 12 bytes of the ciphertext are extracted, to be used as the current Private ID. Note that since only 12 bytes are extracted from the ciphertext, the last two bytes of the byte array in Table 2 is actually not needed to produce the Private ID.

Figure 2 describes the full advertisement payload is then generated using the current Private ID and two other bytes of information.

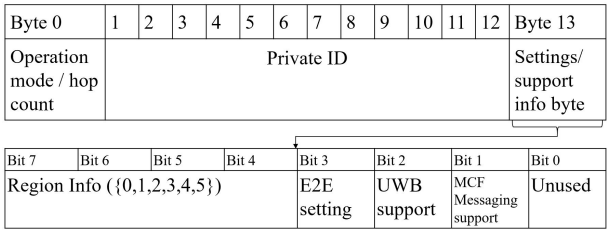

Figure 2: Payload Format for FMM Lost Mode Advertisements

The first byte describes the operation mode of the OF protocol that is being used by a lost device. In the main OF protocol, this byte is always zero. The last byte contains information about the device's region and functionalities supported. This last byte varies depending on the device but stays consistent for all advertisements for a device. If two lost mode devices are advertising in the same area, then this last byte can be used as a quasi-differentiator between the two, provided they do not have the same settings/support.

Once the advertising data has been generated, the lost device starts advertising over BLE on the OF service's UUID FD69. The device will continuously advertise the same data until a timer is triggered that causes it to shuffle the advertising data. This timer is set to trigger every 60 minutes, after which the device generates a new random nonce to be used to generate the advertising data.

Since, there are only 51 possible values for the random, and the random is the only source of non-determinism, there are also only 51 possible values of the advertising data (for a `PrivateIDConfig`). The lost device repeats this process until it is online again. If you have access to the device's secret key and IV (which is standard across all devices), then it is trivial to generate the 51 possible values. It is expected that the Samsung servers store each device's 51 possible values and then performs simple pattern matching to identify a device when required.

3.2.3 Online (Helper) Device Operation. When a device with OF enabled is online, it periodically scans over BLE. This scan is performed with a scan filter that makes sure only advertisements with the OF service's UUID (FD69) are returned from the scan. If there are any lost mode devices nearby, the helper device picks up their advertisements and parses through the data to extract the lost device's private ID. To facilitate multiple lost devices nearby, each helper device maintains a local SQL database in which it adds any lost devices to.

The helper device then stops scanning and starts the location reporting process. Firstly, it accesses the SQL database to get the list of lost devices to report. Then the helper uses any available location services (GPS, Wi-Fi etc.) to pinpoint its own location and record it. The next process differs depending on whether the user has chosen to encrypt their location or not.

- Unencrypted: The helper device creates a HTTPS request containing the lost device's private ID and its unencrypted latitude/longitude. The request is then sent to Samsung's location reporting server.
- Encrypted: The helper device first contacts Samsung (over HTTPS) to receive a public key. Then it creates another HTTPS request containing the lost device's private ID and its latitude/longitude. The latitude/longitude are encrypted using an Elliptic Curve Integrated Encryption Scheme (ECIES) with the public key that was received from Samsung. The request is then sent to Samsung's location reporting server. It is unclear whether the public key from Samsung is standard for all devices or not.

In both cases, the helper device receives a response indicating the success of the operation and the lost device location reporting process finished. The helper device returns to scanning but with a timeout (20 mins) for reporting any lost devices it has already reported.

3.2.4 Device/Account Deregistration. The deregistration process is unclear as most of the operation happens on the Samsung side, however, we can still infer some details based off experimentation and what is evident on the device side. The deregistration process is triggered when OF is toggled off or the FMM application is disabled. This must be done with an active network connection, otherwise the device will not accept the toggle off.

To unregister, the device sends a HTTPS request to the "unRegisterDevice" URL of Samsung's servers. This request contains only the hashed device ID and the request response does not contain any information except for an indication of the operations success. If the response indicates that the operation was successful, then the device clears its lost device database and triggers the Offline Finding service to stop.

From experimentation, we have been able to deduce some properties of the deregistration process. The deregistration does not fully clear the device/account from Samsung's storage. If the same account is logged back into the same device, a new registration process is started. However, the secret key Samsung sends the device remains identical to the previous key for that device/account combination. The secret key was observed to be refreshed only after a different Samsung account was logged in and registered to the device. Reregistering the original account then returns a refreshed secret key.

4 OF PROTOCOL FOR SMARTTAGS

This section discusses the key findings on the Galaxy SmartTag. The proprietary protocols and data exchanges involved in various processes, including registration, Offline Finding, and removal of a SmartTag will be presented in detail. Our analysis was performed on SmartTags with firmware versions 1.01.26 and 1.02.06. Samsung has mentioned to us that they have released a patch to address some of the issues below in the firmware version 1.02.07. At the time of writing, we have not yet performed a detailed analysis to confirm whether all the issues we raised have been patched.

4.1 Methodology

A combination of investigation approaches were used to understand the OF protocol for SmartTags.

4.1.1 BLE Passive Scanning. We used wireshark and the in-built Bluetooth hardware of a research laptop to capture the BLE packets sent between a SmartTag and its owner's device passively, which allowed us to silently observe patterns in the BLE advertising behavior of a SmartTag under different conditions.

4.1.2 Device Loggings. We analysed the runtime behavior of the SmartThings app via device logging using the Android Debug Bridge (adb) over a USB connection. The command line tool `adb logcat` was used for logging system messages produced during the registration and firmware update process of a SmartTag. Some messages contained verbose information, such as data and description, which allowed some of the inner workings and protocols involved in these processes to be inferred. Analysis of the Bluetooth HCI snoop log was an essential approach for us to understand the OF protocol. With the Bluetooth HCI snoop log option enabled, the Android framework will capture the Bluetooth communication between the central and peripherals and store them as a part of the bug report. This allowed the data exchanged between an owner's phone and a SmartTag over BLE connections under different events and operations to be captured and examined.

4.1.3 Android APK Reverse Engineering. We used APK reverse engineering to understand the close-sourced details of the OF protocols for "SmartThing Find". We extracted the APK file of SmartThings using adb. Then we used the JADX decompiler to convert the APK file into Java source code, and performed static analysis on the source code. We identified and understood several functions and classes that are important to the BLE implementation of SmartTags through this method.

4.1.4 Web Traffic Analysis. We used the BurpSuite tool to set up the research laptop as a proxy server to monitor data exchanged between online devices in the OF network and external parties over the internet.

4.1.5 Firmware Reverse Engineering. The dumped firmware of SmartTag [3] was reverse engineered using Ghidra. Analysis of the firmware dump shows that two of the dumped firmware images contain contents of the 512KB flash memory of a SmartTag. Analysis of the firmware has allowed us to recover cryptographic algorithms used in the registration process of SmartTags, which involves ECDH key establishment as indicated by findings from the previous investigation approaches, and SHA256 hashing.

4.2 Overview

The interactions between different parties inside the OF network can be categorized into online interactions (between online devices and remote servers) and offline interactions (between online devices and SmartTags). Topics covered in later sections involve a combination of online and offline interactions.

4.2.1 Online Interaction. Three servers are responsible for the majority of online interactions that will be discussed in the later sections.

- `OAuth Server`: corresponds to samsungosp.com, which provides One SSO Provider (OSP) service for user authentication and authorization.
- `SmartThings Server`: corresponds to client.smartthings.com and api.smartthings.com, which provides web services for various user activities made through the SmartThings application, such as device management operations, e.g., adding/removing a SmartTag.
- `Location Server`: corresponds to chaser-***.samsungiotcloud.com. The `chaser` subdomain is region dependent.

4.2.2 Interacting with the SmartThings Server. The OAuth 2.0 authorization framework is implemented to authorize access to protected resources. When a user signs into the SmartThings application:

- the app will make a POST request to the `/auth/oauth2/requestAuthentication` URL of the `OAuth Server` to obtain a signin token `userauth_token`.
- Then, the app will make a POST request to the `/auth/oauth2/authWithTncMandatory` URL of the `OAuth Server` to obtain a Bearer token `access_token`, which authenticates the user to the `SmartThings Server`.

Any interactions with the `SmartThings Server` require a valid Bearer Token (`access_token`) to be present to grant the requester access to its web services.

4.2.3 Interaction with the Location Server. Online devices in the OF network communicate with the Location Server via HTTPS. The owner's device of a newly registered OF device would make a location report to create a new OF device profile on the Location Server; a helper device would report found lost devices to the location server to allow the owner receive location updates of the lost device.

Interactions with the `Location Server` uses a different authentication scheme. It requires a valid JWE (JSON Web Encryption) token to be present in the request header to authenticate the requester to the server. Each new authentication token is obtained by interacting with URLs of the `Location Server` through the following procedure:

Nonce request. The client makes a GET request to the `/nonce` URL to obtain a 16-byte nonce randomly generated by the server. We note here that the nonce is generated by the server so the client has no control over its value.

Access token request. Then, the client makes a POST request to the `/accesstoken` URL of the `Location Server` to obtain a new JWE access token.

The request headers contains a `certificate`, `signature`, and a `nonce` field. The value for those fields are formed by two certificates and a private key are loaded from a keystore file on the client's device's file system using the nonce received during the last step.

certificate The value of this field is a certificate chain, consisting of two X.509 certificates. The first certificate is an intermediate CA certificate, which is used to sign the 2048-bit RSA public key (for signature verification) in the second certificate.

signature The value of this field is produced by encrypting the SHA-256 hash of the `nonce` with the private key that corresponds to the RSA public key in the second certificate above.

nonce the nonce used to generate the signature

The server's response for a valid request would contain a new authentication token that stays valid for 32 hours.

Location report. The received access token is present in each location report request to the server. Apart from information about the SmartTag and the access token, each location report contains an *id* field, generated as follows:

$$id = androidId[0:4] \| SHA256(androidId \| \text{“}findMyMobile\text{”})$$

where *androidId* denotes the Android Device ID of the helper device.[10] Since Android 8.0, the Android Device ID is unique to each combination of the signing key of the app (SmartThings in this case), user and the device, whereas prior to Android 8.0, it is a static identifier unique to each device (but may change when the phone is factory-reset).

4.2.4 Offline Interaction. Offline interactions use BLE, which has two ways for data transmission, the first is through BLE advertisement, and the second is through data exchange during an established connection using the GATT server.

BLE Advertisement. A SmartTag broadcasts BLE data that any nearby online device can pick up. A non-registered tag broadcasts on UUID FD59, which allows an online device to discover its presence before the registration procedure. A registered tag broadcasts a 20-byte payload that follows a fixed structure on UUID FD5A, which allows the tag to participate in the OF network.

[10] https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID

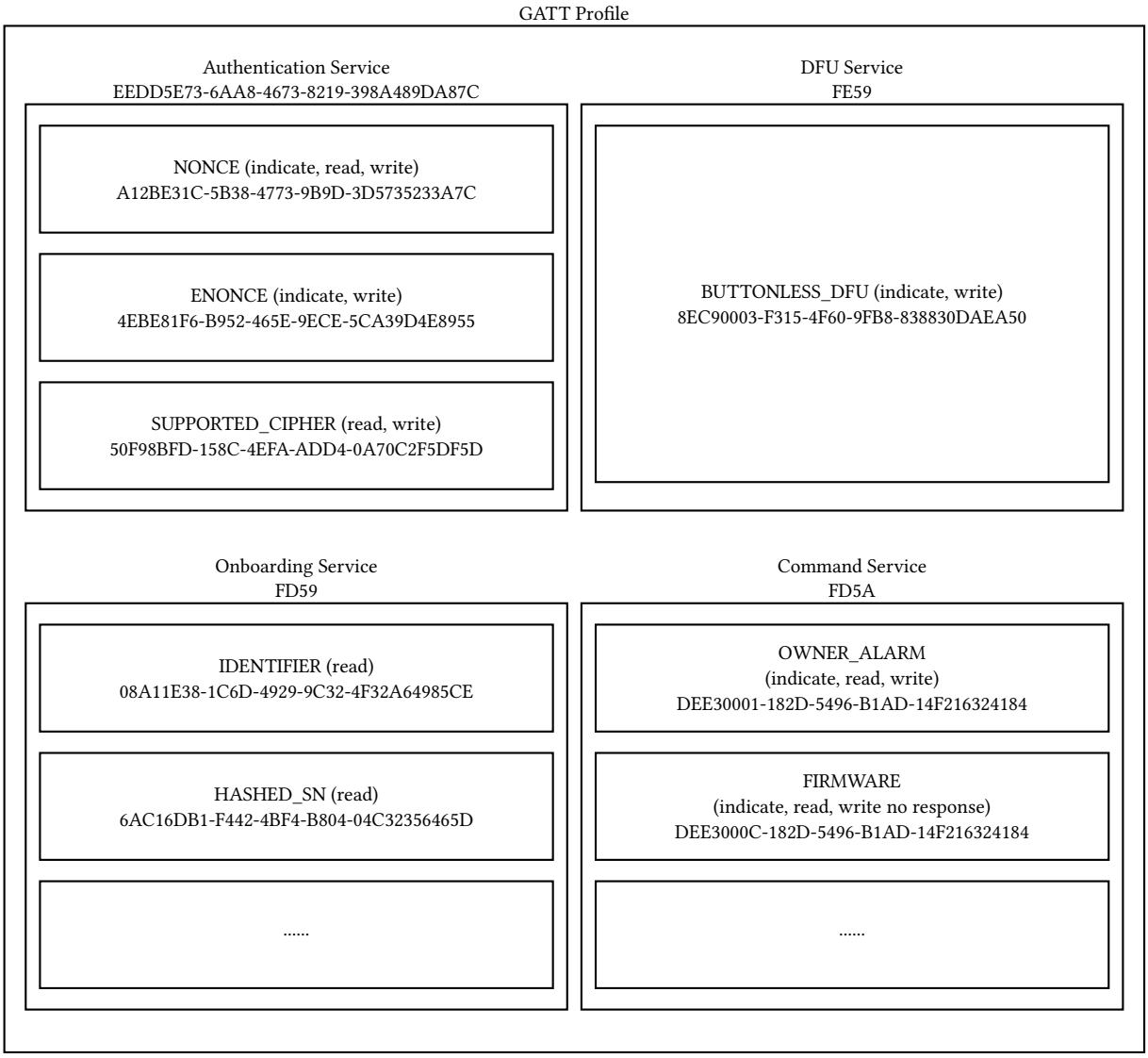

Figure 3: The GATT Server Profile for SmartTags

GATT Interaction. A SmartTag has an active GATT server for data exchange over connections. Figure 3 provides an overview of its architecture. The service has four primary services which can be summarized as follows:

Authentication Service The Authentication Service for SmartTags has the same UUID and characteristics as for FMM phones.

DFU Service Service UUID FE59 is a part of the nRF52833 Buttonless Secure DFU service for over-the-air firmware updates.

Onboarding Service Service UUID FD59 is used for device onboarding/registration activities. During the registration process of a SmartTag, the owner device and the tag would exchange configuration and cryptographic data over various characteristics under this service.

Command Service Service UUID FD5A is primarily used for performing more complicated interactions between the owner device and a tag, such as executing a supported command (e.g., alarm, changing ringtone) on the tag through data exchange on the corresponding characteristic in the Command Service.

4.3 SmartTag Registration

The registration process of a SmartTag involves offline interactions with the tag (Owner-Tag), and online interactions between the owner's device and URLs under the smartthings.com domain, and the `/geolocation` URL of a `chaser` subdomain (§4.2.1) (Owner-Server). The registration process is managed by the SmartThings application, where a user needs a Samsung account to log in. A

Table 3: Format of a non-registered SmartTag's advertisement

Byte(s)	Description	Value
0		01
1-4	manufacture ID (mnId)	30414644 (0AFD)
5-7	setup ID (setupId)	343330 (430)
8-10		010501
11-13	last two bytes of the MAC address	varies, e.g., 33444431 for 3D:D1

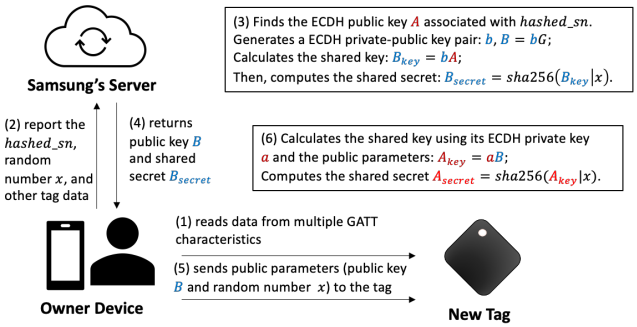

Figure 4: Shared secret establishment

user can use the "add a device" option in Smartthings to start the registration process.

The user's phone will perform a BLE scan to detect nearby non-registered SmartTags. A non-registered SmartTag broadcasts static BLE data unique to each tag on UUID FD59. Table 3 details the advertisement structure of an non-registered tag. The registration flow can be devided into six stages, which will be explained next.

4.3.1 Shared Secret Establishment. During the registration process, a shared secret is established between the user and the tag to secure subsequent communications. Computation of the shared secret involves ECDH on curve25519 and SHA-256 hashing. The interactions between each party (Owner, Server, Tag) are summarized in Figure 4.

request (steps 1-2) The phone makes a POST request to the `/identity/easysetup/blob` URL of the clientsmartthings.com domain.

```
{
    "keyid": {"type":"hashed_sn","value": hashed_sn},
    "mnId":"0AFD",
    "rand": x,
    "setupId":"430"
}
```

The listing above shows the body of the `shared secret request`, which consists of the following information collected from the tag's GATT characteristics and BLE advertisements:

general information The values of `mnId` and `setupId` were obtained from the BLE advertisement data of the tag.

information unique for each tag The `hashed_sn` is the base64 encoding of tge first 6 bytes of the hashed serial number read from the `HASHED_SERIAL_NUMBER` characteristic. The random number `x` is a 32-byte value randomly generated by the phone for each registration session. Those two parameters are the public parameters for the shared secret establishment process.

Table 4: Characteristics for exchanging public parameters

Name	UUID	Value
CLOUD_PUBLIC_KEY	b5754629-6821-44c6-a118-492feecf6bb2	32-byte, varies
RANDOM_VALUE	6ac16db1-f442-4bf4-b804-04c32356465d	32-byte, varies

response (steps 3-4) The server returns an HTTPS response containing the shared secret B_{secret} and the public key B_{pub} to the phone: after receiving the request, the server will find the public key of the tag A_{pub} associated with the `hashed_sn`. Then, the server will generate an ephemeral private-public key pair b and $B_{pub} = bG$, and computes the ECDH shared key. Finally, the shared key is concatenated with the random number `x` to form the input for the SHA-256 hash function to produce the shared secret: $B_{secret} = SHA256(B_{key}|x)$.

Steps 5-6 After receiving the response, the phone sends the ephemeral public key B_{pub} and the random value x to the tag via Write Requests to the corresponding GATT characteristics (see Appendix 4), allowing the tag to compute the shared secret in the same manner.

Deriving AES keys from the shared secret. The first 16 bytes of the shared secret are taken as the `masterSecret`. It is used to derive four sub-keys for securing different OF-related cryptographic operations in subsequent communications between the phone and the tag. Note that Samsung OF protocol does not use any of the default BLE pairing and authentication mechanisms, so this encryption key is unrelated to BLE Long Term Key (LTK) that is normally exchanged as part of BLE pairing protocols [11].

- **authKey**: This key is used by the owner to establish an authenticated BLE session with a SmartTag. It is computed by taking the first 16 bytes of the following SHA256 digest:

$SHA256($

Bytes 0-15	Bytes 16-19	Bytes 20-N
masterSecret	00000001	"smartthings"

$)$

- **gattKey**: This key is used for encrypting the data exchanged in the GATT interactions between the phone and the tag. It is computed by taking the first 16 bytes of the following SHA256 digest:

$SHA256($

Bytes 0-15	Bytes 16-19	Bytes 20-N
masterSecret	00000001	$nonce_{tag}$

$)$

The $nonce_{tag}$ is a 16-byte value received from the SmartTag during each BLE authentication process, which will be discusses §4.3.1.

- **pidKey**: This key is used for generating unique identifiers (caleld Privacy IDs) for a SmartTag to broadcast when it is in a lost mode (see §4.5). It is computed by taking the first 16 bytes from the following SHA256 digest:

$SHA256($

Bytes 0-15	Bytes 16-19	Bytes 20-N
masterSecret	00000001	"privacy"

$)$

- **signKey**: This key is used for signing and validating the integrity of the BLE data broadcasted by a SmartTag, and is

generated from the first 16 bytes of the following SHA256 digest:

$SHA256($

Bytes 0-15	Bytes 16-19	Bytes 20-N
masterSecret	00000001	"signing"

$)$

The usage of each key will be discussed in related sections.

BLE Authentication. After computing the `masterSecret`, the owner device will initiate a two-way authentication with the tag to establish an authenticated connected session with the tag using the *authKey* derived from the `masterSecret`. The Authentication Service (see §4.2.4) is responsible for data exchange during the authentication procedure. This service contains three characteristics, which are the same as the characteristics used in the authentication service for FMM (see Table 1).

phone → tag: The owner's phone sends a randomly generated 16-byte $nonce_{owner}$ to the `NONCE` characteristic of the tag via Write Request.

tag → phone: The tag responds with a randomly generated 16-byte $nonce_{tag}$ via Indication on `NONCE` characteristic.

phone → tag: The phone encrypts the string "smartthings" using $nonce_{tag}$ as the IV, and then writes it to the `ENONCE` characteristic of the tag via Write Request.

tag → phone: The tag responds with an encryption of the same string "smartthings" but with $nonce_{owner}$ as the IV, via Indication on `ENONCE` characteristic.

The encryption in the third and the fourth steps above use the AES/CBC/PKCS7 cipher:

$$E_{authKey}(\underbrace{nonce}_{\text{IV}}, \underbrace{\text{"smartthings"}}_{\text{Plaintext}})$$

The authentication succeeds if the correct encrypted value is received by both party.

4.3.2 Securing GATT interactions. Completion of the BLE authentication allows the *gattKey* to be derived using the `masterSecret` and $nonce_{tag}$, as described in §4.3.1.

The *gattKey* is used for encrypting sensitive data exchanged between an owner device and a tag within an established authenticated GATT connection. Characteristics responsible for exchanging more sensitive information, e.g., characteristics with the "encrypted" property that will be discussed in later registration stages, require the data to be encrypted by using the AES/CBC/PKCS7 cipher before being exchanged. The cipher uses the *gattKey* as the encryption key, and the $nonce_{tag}$ as the IV, to encrypt/decrypt raw data (*raw*) exchanged over these characteristics:

$$encryptedData = E_{gattKey}(\underbrace{nonce_{tag}}_{\text{IV}}, \underbrace{raw}_{Plaintext})$$

The structure of the raw data will be elaborated in §4.4.

4.3.3 Ensuring Physical Ownership of the Tag. When registering a tag through the normal flow, the SmartThings app will ask the user to press the tag button to ensure physical ownership of the tag. This stage only involves BLE communications between the owner's device and the tag.

Pressing the tag button sets the value of the `CONFIRM_STATUS` characteristic to 0x01 (encrypted) from the default value 0x00. The owner's device would only continue the registration flow after validating the value of this characteristic.

Name	UUID	Value	Encrypted
CONFIRM_STATUS	f299f805-17b3-43c1-ac12-fbcc59ee2f0d	0x01	Yes

4.3.4 Ownership Status Check. This stage checks the ownership status of the tag to ensure that it is not currently registered to another user.

request The owner's device makes a GET request to the `/chaser/trackers/lostmessage` URL of the client.smartthings.com domain with the `serialNumber`, `modelName`, `mnId`, and `setupId` GET parameters provided, to check the ownership status of the tag.

response The Status code of the response is generally either 200 or 404:

- 404 means the profile does not exist. This should be the response for a non-registered SmartTag, as the tag's parameters should not correspond to any existing profile on the location server.
- 200 means a profile that matches the parameters exists. A 200 response body should contain an `own : Boolean` field. **`true`** means the tag is currently registered to the requester. **`false`** means the tag is registered to another user.

Thus, the registration process would only proceed if the response status code is 404, or 200 and `own==`**`true`**.

4.3.5 Finalizing Tag Registration. This stage creates an online profile of the tag that associates with the owner's SmartThings account.

request The owner's device makes a POST request to the `/miniature/mobile` URL of the client.smartthings.com domain.

```
{
    "tagData":{
        "firmware":{
                "specVersion":"0.5.3",
                "version":"01.01.26"
            },
            "mnId":"0AFD",
            "modelName":"EI-T5300",
            "serialNumber": sn,
            "setupId":"430"
        },
    "cipher": "AES_128-CBC-PKCS7Padding",
    "configurationVersion": "2.0",
    "identifier": sn,
    "deviceName": ...,
    "encryptionKey": shared_secret,
    "locationId": ...,
    "mnmn": "Samsung Electronics",
    "roomId": ...,
    "vid": "IM-SmartTag-BLE"
}
```

The listing above shows the body of the `finalization request`, which consists of the following information:

Table 5: Characteristics read for making the `finalization request`

Name	UUID	Value	Encrypted
SPEC_VERSION/specVersion	dee3000e-182d-5496-b1ad-14f216324184	"0.5.3"	No
FIRMWARE_VERSION/version	30c48d2a-6ccb-4240-9f97-7f97a3f1c030	"01.01.26"	No
MODEL_NAME/modelName	d19ddd83bbe14144bb18f3ceb57c480a	"EI-T5300"	No
SUPPORTED_CIPHER/cipher	5b5f7a4c-257e-4841-92d5-0042658122b6	"AES_128-CBC-PKCS7Padding"	No
CONFIGURATION_VERSION/configurationVersion	12761292-241c-490c-8424-6f7cc8a8a027	"2.0"	Yes
MNMN/mnmn	04052818-d201-43eb-9d81-e936dc86ee06	"Samsung Electronics"	Yes
VID/vid	77b08bec-5890-49d1-b021-811741b417e6	"IM-SmartTag-BLE"	Yes

general information The values of `mnId` and `setupId` were obtained from BLE advertisement data as shown in Table 3. The values colored in blue were read from the corresponding GATT characteristics under the onboarding service (UUID FD59) (see Appendix 5 for details).

requester specified The `deviceName` value is "SmartTag" by default. It can be any custom value specified by the owner's device and determines the name of the registered device displayed in the SmartThings application. The `locationId` and `roomId` values are keys for users to access their registered devices stored in a specific room of a location. Each new account has a default location and two default rooms, each associated with a unique and static id. Devices registered to an account is accessed in a `locationId -> roomId -> devices` way.

unique for each tag The values of `serialNumber` and `identifier` equate the identity MAC address of the tag. The `encryptionKey` field contains the shared secret received from the earlier stage (§4.3.1).

response The server's response to this request contains configuration data associated with the tag shown in Listing 1.

Listing 1: The `finalization response` Body

```
{
    "deviceId":...,
    "metadata":
    {
        "regionCode":...,
        "privacyIdPoolSize":...,
        "privacyIdSeed":...,
        "privacyIdInitialVector":...,
        ...
    }
}
```

The `deviceId` value is used to access the profile of the tag for various operations, e.g., removing the tag, pulling the location history of a lost tag.

The four values contained in the metadata are sent to the tag via Write Requests to the corresponding GATT characteristics for the tag to generate BLE data for OF after completing the registration process (see Table 6 for details). Finally, the phone will perform time synchronization with the tag through the `TIME_SYNC` characteristic, then write a value to the `SETUP_COMPLETE` characteristic to indicate the completion of the registration process (see Table 8 and Table 7). The tag will then drop the GATT connection with the owner device to operate in the registered mode and broadcast OF data on RPAs.

4.3.6 Setting up OF Device Profile on the Location Server. After completing the above stages, a device profile of the tag is added to the owner's SmartThings account. The owner's device with FMM enabled would then scan BLE to discover its registered tag through the privacy ID in its BLE data.

After recognizing the registered tag, the owner device would create a POST request to the `/geolocations/`*deviceId* URL of the `Location Server` to create an OF device profile of the tag on the server. The authentication mechanisms involved in the server communication process was discussed in §4.2.1 .

This profile bonds the ownership status of the tag with the owner's Samsung account, which prevents the tag from being registered by others, as discussed in §4.3.4.

4.4 Owner-Tag GATT Interaction

When an owner device is in-range with its registered SmartTag, the SmartThings application will automatically initiate the BLE authentication with the tag using the procedure described in §4.3.1. After successfully establishing an authenticated connection, the application will display the status of a SmartTag as connected. An owner can then perform various supported actions with a connected tag through SmartThings, such as setting off the alarm on the tag.

Each action is triggered by an exchange of data that encodes commands related to the action. Concretely, these exchanges are done through a GATT characteristic under the Command Service (UUID FD5A) of the SmartTag. The data is encrypted using the *gattKey* derived during the authentication procedure (see §4.3.1). Recall that *gattKey* is dependent on the nonce that the tag provided during authentication, so the *gattKey*, under normal operations, is unique to each connection.

4.4.1 Command Structure. Commands are data being exchanged between an owner device and the tag for triggering a specific action on the receiver. Commands can be sent through characteristic Write Requests (owner to tag) or characteristic value indication (tag to owner). The structure of a command is as follows:

$Data =$

Bytes 0-3	Byte 4	Bytes 5-N
Counter	Opcode	Argument(s)

Bytes 0-3 store a counter `counter`, encoded in little-endian format. The value of the `counter` corresponds to the total number of GATT commands the device has successfully sent during the

Table 6: Characteristics written to set up a tag for OF

Name	UUID	Value	Encrypted
REGION	bebfaa51-dcb8-44de-a4b8-fc8c9c7ef46d	a valid region code, e.g. 12 for AU	Yes
Privacy ID Seed	d19ddd83-bbe1-4144-bb18-f3ceb57c480a	12-byte, varies	Yes
Privacy Pool Size	7534c394-1f40-4d12-afd7-dc2a75bd6a44	1000	Yes
privacy ID IV	abd6e6ba-3843-4786-b9b2-b69548eed881	16-byte, varies	Yes

Table 7: The `SETUP_COMPLETE` characteristic

Name	UUID	value	Encrypted
SETUP_COMPLETE	bcc8cce6-8af6-48dc-a0ae-547f7c095229	"FINISH"	Yes

Table 8: The `TIME_SYNC` characteristic

Name	UUID	Value	Encrypted
TIME_SYNC	dee30005-182d-5496-b1ad-14f216324184	Will be set to the current UTC Time	Yes

current authenticated session. Byte 4 is the command opcode indicating the type of command the data contains, as each characteristic in the Command Service may have multiple supported commands.

Commands that are exchanged between the owner and the tag are encrypted with the *gattKey* using AES/CBC/PKCS7 cipher. When an encrypted command is received, the receiver decrypts it to obtain the plaintext command, and validates the command before executing it. However, our reverse engineering suggests that there is a slight difference in how the owner and the tag validate the commands they receive. The tag would proceed to execute the command if the opcode and the argument(s) are in the expected form, but it ignores the counter, wherease the owner device additionally performs a simple validation of the counter. The owner device uses a global variable (*max*) to store the largest counter of the commands received so far in the current authenticated session. After decrypting a command with counter n received from the tag, the owner would compare the value of *max* with n. If n is greater than the value of *max*, then owner updates *max* with n and proceeds to execute the command. Otherwise, the command is discarded.

We describe some interesting commands that we have identified.

Playing Sound on SmartTags. The GATT characteristic `OWNER_ALARM_CHAR` (UUID DEE30001-182D-5496-B1AD-14F216324184) is used to configure the alarm on a tag. A phone can make a SmartTag ring by sending a command packet, with the command opcode 0x01, through this characteristic. The opcode for turning off the alarm is 0x00.

Remote Ring. The Remote Ring feature allows a tag to make its owner device ring by pressing the tag button. This feature is not enabled by default and needs to be explicitly enabled by the owner through the SmartThings app. The owner device can receive this command through subscribing to the indication of the `remoteRing` GATT characteristic (UUID DEE30003-182D-5496-B1AD-14F216324184) of the tag.

A tag's button has three states, which are `pushed`, `held`, and `pushed_2x`. The Remote Ring is triggered by double-pressing the tag button. After the first press, the Button Characteristic will send an indication with the value 0x01, which corresponds to `pushed`. After the second press, the characteristic will send another indication with the value 0x03, which corresponds to `pushed_2x`.

When the owner device receives 2 Button Characteristic indications, where the second indication has a value of 0x03 and a greater counter value than the first indication, the remote ring will be triggered.

Firmware Update. The vendor has implemented a custom channel for performing firmware updates on SmartTags. Over-the-air firmware updates can be performed by writing encrypted commands to the `FIRMWARE_CHAR` (UUID DEE3000C-182D-5496-B1AD-14F216324184). After establishing an authenticated session. The SmartThings application will automatically read the firmware version of a tag from characteristic UUID DEE3000B-182D-5496-B1AD-14F216324184. If the firmware version is below the latest version, the application will ask the user to start a firmware update. Once the firmware update is triggered, the SmartThings app would download the latest firmware through an API call, which would generate a temporary link (that would expire after a couple of hours) to download the firmware (from the domain smart-tag-firmware.samsungiotcloud.com). The downloaded firmware will be cached locally.

To initiate a firmware update on the tag, the phone must first subscribe to the `FIRMWARE_CHAR` characteristic's indication, then execute the `transferFirmwareInformation` command by writing opcode 0x00 and a list of arguments to the characteristic, as shown in Table 9. The value of the `transferWindow` argument is 0x0A (10) in the BLE packets we observed, which means that the tag will send a handle value indication after receiving every ten firmware data packets and wait for the phone to confirm.

By analyzing the device logs produced during the update process, we could observe all the firmware update commands and

Table 9: Format of the transferFirmwareInformation command

Opcode	Argument	Type
0x00	totalFirmwareSize	uint32
	totalFirmwareCRC16	uint16
	newFirmwareVersionLength	uint8
	newFirmwareVersion	string
	transferWindow	uint8

arguments being executed. Table 10 shows the arguments for the `transferFirmwareInformation` command, which indicates that the firmware has a total size of 179620 (0x2bda4), total CRC of 37858 (0x93e2), firmware version 1.02.06 (0x312e30322e3036), and transfer window size of 10 (0x0a).

Table 10: Captured transferFirmwareInformation command

Byte	0	1-4	5-6	7	8-14	15
Value	00	a4bd0200	e293	07	312e30322e3036	0a

Then the phone will start to execute the `transferFirmwareData` command by writing opcode 0x01, followed by a list of arguments including segmented firmware data to the characteristic until all the firmware data is transferred. Table 11 shows the format of the `transferFirmwareData` command.

Table 11: Format of the transferFirmwareData command

Opcode	Argument	Type
0x01	offset	uint32
	segmentedFirmwareDataLength	uint16
	segmentedFirmwareData	byteArray
	argumentsCRC16	uint8

4.5 The offline-finding protocol for SmartTags

There are three types of devices participating in the location tracking process of a lost SmartTag: the Owner Device, the Lost Device (tag), and the Helper Device. Figure 5 provides a high-level overview of the lost-and-found process, which involves three types of interactions:

- Helper-Tag: A lost tag broadcasts OF data that is available to any nearby Bluetooth devices, while a Helper Device periodically scans for lost tags over BLE, and picks up the OF data from a nearby lost tag.
- Helper-Server: After receiving the OF data, the Helper Device would report relevant information of the found lost tag to Samsung's server.
- Owner-Server: The Owner device can hence download the updated location of its lost device from Samsung's server.

The present section details the OF protocol for SmartTag based on the type of interactions involved in the protocol.

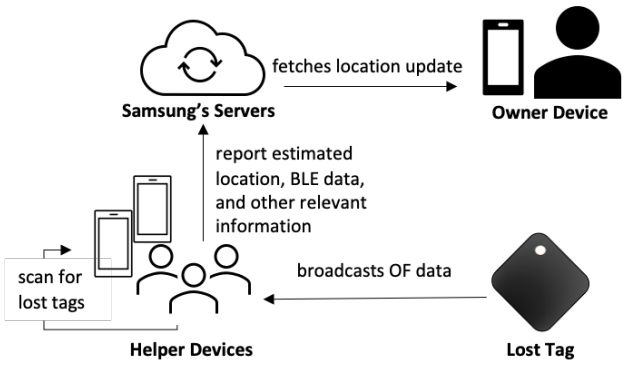

Figure 5: The lost and found process of a SmartTag

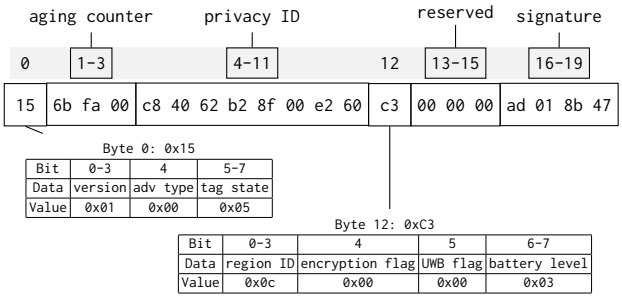

Figure 6: The OF Advertisement Structure for SmartTags

4.5.1 Helper-Tag Interaction. A registered tag broadcasts OF advertisements on UUID FD5A continuously. Any active Galaxy device with "FindMyMobile" (FMM) enabled is a helper device that participates in Samsung's OF network. A helper device regularly scans for BLE advertisement data from nearby SmartTags. It filters BLE advertisements based on the advertising UUID for SmartTags (FD5A). A helper device does not attempt to make any GATT connection to the lost tag, so all relevant information about the lost tag must be encoded in the advertisement data, which is detailed next.

Advertisement Structure. Figure 6 shows the advertisement structure of the OF data broadcasted by a SmartTag. Byte 0 stores the tag version, advertisement type, and tag state. Bytes 1-3 are the aging counter. Bytes 4-11 correspond to the first 8 bytes of a privacy ID selected from the privacy ID pool of the tag, which uniquely associates the tag with the Samsung account it is registered to. Byte 12 stores the region code, encryption flag, ultra-wide band (UWB) flag, and the battery level of the tag. Bytes 16-19 store the 4-byte signature for validating the integrity of the BLE payload.

Tag State (Byte 0) Bits 5-7 of Byte 0 in the OF data of SmartTags store the operating state of a tag. There are six different tag states, inferred through a combination of reverse-engineering and BLE scanning.

The state of a registered tag becomes Premature Offline once it is disconnected from its Owner Device or rebooted. After staying disconnected for 15 minutes, the tag state would changes to Offline Mode, which would inform nearby Helper Devices that

Bits 5-7	Name	Description
001 (1)	Premature Offline Mode	the tag has recently been disconnected
010 (2)	Offline Mode	the tag has remained disconnected for over 15 minutes
011 (3)	Overmature Offline Mode	the tag has stayed in Offline Mode for over 24 hours
100 (4)	Paired with one device	the tag is paired to a device
101 (5)	Connected to one device	the tag is connected to a device
110 (6)	Connected to two devices	the tag is connected to two devices

Table 12: Operating States of a SmartTag

the tag is considered lost. After operating in Offline Mode for 24 hours, the tag state would change to Overmature Offline Mode. A SmartTag in the Overmature Offline mode would slow down certain computation process for power saving, which will be discussed later in the section.

Helper devices in Samsung's OF network will only report locations of SmartTags in Offline or Overmature Offline mode.

Aging Counter (Byte 1-3) Bytes 1-3 are used to store the aging counter, which can be seen as the timestamp for the BLE data. The formula for a SmartTag to generate an aging counter is $agingCounter = (tagTime - 1593648000)/900$. Here the constant 1593648000 is an encoding of timestamp in Unix Epoch Time, which in this case equals July 2, 2020, 12:00:00 AM GMT. So the aging counter can be understood as the elapsed time since July 2, 2020, 12:00:00 AM GMT, divided by 900 seconds (15 minutes). The aging counter should therefore be the same for all SmartTags in sync with the server time and changes every 15 minutes.

Privacy ID (Byte 4-11) Bytes 4-11 are used to store an 8-byte privacy ID, which is an unique identifier of a SmartTag. Each registered SmartTag has a set of unique privacy IDs called the privacy ID pool. The pool is deterministically generated using the $pidKey$ and the privacy ID configurations ($privacyIDIV$, $privacyIDSeed$, $privacyPoolSize$) that the server sent to the owner during the finalization stage in the registration process (§4.3.5). The privacy ID is an essential part of SmartTag OF protocol as it associates a tag with its owner's Samsung account. The AES/CBC/PKCS7 cipher is used to generate the privacy ID pool of a SmartTag. The cipher is initialized with the $pidKey$ as the encryption key and the $privacyIDIV$ as the initialization vector. The privacy ID pool generation function uses a for-loop that iterates $privacyPoolSize$ times, where each iteration forms a unique input that is encrypted by the cipher to produce a new privacy ID and adds it to the pool:

$$pid_i = E_{pidKey}(\underbrace{privacyIdIV}_{\text{IV}}, \underbrace{input_i}_{PlainText})$$

where $i \in [1, \ldots, privacyPoolSize)$ and $input_i$ is

Byte 0	Byte 1	Bytes 2-9	Byte 10	Byte 11
$i \gg 8 \wedge 256$	$i \wedge 256$	privacyIdSeed	$(i \gg 8) \wedge 256$	$i \wedge 256$

.

Each input is a 12-byte value generated by the privacy ID seed and the current iteration index i. Byte 0 of the array is produced by Right-Shifting i by 8 bits, then ANDing it with 256. Byte 1 is produced by ANDing i with 256. Bytes 2-9 are the 8-byte privacy ID seed. Byte 10 is a copy of byte 0, and byte 11 is a copy of byte 1. The privacy ID field in the advertisement data uses first 8 bytes of a privacy ID. The privacy ID field in the advertisement data uses first 8 bytes of a privacy ID. The privacy pool size for SmartTags is 1000 (for the firmware version we analyzed), which is larger than that for FMM.

Signature (Byte 16-19) The signature field at the end of the BLE data serves as a cryptographic checksum for the first 16 bytes of the BLE advertisement data of a registered SmartTag, which allows an authorized party to validate the integrity of the advertisement. Let $blePayload$ denote the first 16 bytes of the BLE advertisement data. Then the four signature bytes are obtained from the first 4 bytes of the $fullSignature$ defined below:

$$fullSignature = E_{signKey}(\underbrace{privacyIV}_{\text{IV}}, \underbrace{blePayload}_{\text{Plain Text}})$$

where $E_{signkey}$ denotes the AES/CBC/PKCS7 cipher with the secret key $signkey$ that was derived from the master secret during the tag registration (see §4.3). Therefore, any changes to the first 16 bytes of the advertisement will likely cause the signature bytes to change correspondingly, which allows the integrity of the BLE data to be validated by parties with the privacy ID configuration and the shared secret of the tag, such as the owner and the server.

Advertisement Update. A SmartTag updates its BLE Data periodically. A tag that operates in any non-Overmature Offline state updates its privacy id, aging counter, and signature every 15 minutes: a new privacy ID is selected from the privacy ID pool, and the aging counter increments by 1. Then, the tag recomputes the signature bytes based on the updated BLE payload. Under the Overmature Offline state, a tag updates the aging counter and signature every 15 minutes. However, the frequency for shuffling the privacy ID reduces from every 15 minutes to once every 24 hours.

4.5.2 Helper-Server Interaction. A Helper Device stores found lost SmartTags in a local database together with other lost FMM/FME devices discovered by the Helper. The database can store a maximum of 1000 devices using the privacy ID of the device as the key. A tag is marked as expired if it has not appeared in the BLE scanning for 15 minutes and will be removed from the database.

The helper device will report geolocations of lost SmartTags in the database based on estimated locations received from the WiFi or GPS service. Each request is similar to the one made by the Owner Device to create an OF device profile (see §4.3.6), except that the URL is `/geolocations`, as a Helper Device does not know the `deviceId` of the lost tag. Each location report task allows a maximum of 5 recently found devices ($time_{found} \geq time_{current} - 1$ (minute)) from the local database to be reported.

4.5.3 Owner-Server Interaction. An Owner Device receives location updates of its lost SmartTag through POST requests to an URL under the api.samsung.com domain.

Figure 7 shows an example of a location history request, with some sensitive information redacted (e.g., those marked with X's and the UUID for the `deviceId`). The request is authenticated using the bearer token that the owner device obtains after signing in to SmartThings (see §4.2.1). It also contains the information about the device Id (in the `extraUri` field in the parameters). For this example request, the server would return a list of a maximum number of 200 location reports received between 1662991200001

and 1663077599999 (GMT time) for the SmartTag associated with a `deviceId`.

4.6 SmartTag Removal

A Registered SmartTag can be removed by its owner through SmartThings. If the target tag is connected to the Owner Device, the Owner Device will first perform a factory reset of the tag through GATT interaction by writing the `reset` command (opcode 0x01) to the `tag.factoryReset` characteristic (UUID dee30006-182d-5496-b1ad-14f216324184) under the Command Service.

The Owner Device will also make an DELETE request to the `/devices/[deviceId]` URL of the api.smartthings.com domain to remove the online profile of the SmartTag. After the online profile of a tag is removed, GET request to the /chaser/trackers/lostmessage?serialNumber=...&modelName=...&mnId=...&setupId=... URL would receive a 404 Not Found error response from the server, indicating that no existing profile associated with the device defined by the serial number, manufacture ID, and setup ID is found.

5 SECURITY AND PRIVACY ANALYSIS

We now examine security and privacy issues arising from Samsung OF systems, covering mobile devices (running the FMM app) and SmartTags. We organise our analysis on various subprotocols involved in the interactions between different parties involved in the overall OF system. Then in Section 6 we discuss how these individual issues contribute to our overall analysis of the security and privacy risks of the OF system along the line of the research questions we posed in the introduction (RQ1-RQ4).

5.0.1 Helper Device - Offline Device Interaction.

Privacy ID Pool Size. As discussed previously, the privacy ID pool size for OF devices is relatively small (51), meaning that is not overly difficult to collect the pool of privacy ID values for a device. Experiments have shown that by capturing BLE advertisements from passively scanning, the entire pool of privacy IDs can be collected within a few days.

This vulnerability defeats the privacy feature for FMM devices, which depends on frequent update of identifiable information (privacy ID) in the BLE data and the BLE MAC address of a device to prevent long-term identification from nearby BLE capable devices. An adversary can collect the privacy ID pool of a nearby lost device and permanently identify it by pattern matching the privacy ID contained in its BLE advertisement with the pirvacy IDs in the collected pool.

A simple mitigation method is to increased the pool size. Although this may create scalability issues for Samsung due to the increased data storage requirements for storing the privacy ID pool of each device. Another approach is to use a different cryptography algorithm that allows each privacy ID to be derived deterministically and synchronized between the owner and the tag, instead of setting a fixed pool size and storing a pre-computed pool of IDs for each device.

Replay Attacks. The OF protocol does not require a connection between a lost device and a helper device, in order to report the lost device's location. The helper device only needs the lost device's private ID contained in its advertisements. This makes the OF protocol extremely susceptible to replay attacks. All an adversary needs to do is acquire a registered OF device's lost mode advertisements, and then they can replay the advertisement using a BLE capable device (not necessarily a Samsung mobile) and the advertisement will be picked up by any nearby helper OF Samsung devices. The adversary must make sure that the advertisements created use the OF service UUID (FD69) and then the advertisements will easily pass through the helper OF device's scan filters. The helper device then parses through the advertisement data as normal, extracting the private ID and reporting it back to Samsung. Samsung's servers will then update the location for the registered OF device based off where the spoofed advertisements are, even if the actual registered OF device is somewhere else completely.

An adversary can record the OF advertisements observed from a device to replay them at different locations of their choosing. Any Helper Device at that location would generate a location report for the replayed data and send it to Samsung. The adversary could repeatedly do this at many different locations, with many different advertisements, to confuse the device owner with incorrect locations. This would effectively create a denial of service as OF users would receive confusing location reports, making it impossible to locate their device.

A possible attack scenarios is that an adversary has just physically stolen a device from a victim. In the normal case, the victim could log onto the FMM web service and receive an accurate location report for their device, allowing them (or police services) to track down the thief. However, the thief could use a network of BLE transceivers (or just one) to create a false location trail, leading the victim in the completely wrong direction and making it significantly easier to steal the device without being caught.

This flaw would also be extremely beneficial for an adversary who would like to create a tracking device but does not have their own location network. The adversary simply needs to have a legitimate Samsung device and account, which is easy to do and does not require identification. They would then register their legitimate device with OF, observing the advertising data the device produces, or just calculating the values themselves which would be trivial. They could then use an ESP32 (or similar) device to start BLE advertising with their legitimate device's data. This would give them a small tracking device that utilises the OF network. To access the location of the tracking device, they need only log into the OF web service with their legitimate account. The only concern for the adversary would come from the tracking device being found and linked to their legitimate account, but if they did not provide any identifying information when registering then it would be significantly harder to identify them.

A mitigation method is to add extra fields to the BLE advertisement to ensure the timeliness and integrity of each advertisement. For instance, the `aging counter` and `signature` fields in the advertisement data of SmartTags are used to ensure those two properties. The same implementation can be applied to the rest of the FMM devices in the OF network to limit the effectiveness of the replay attacks for OF advertisements.

```
POST /installedapps/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/execute HTTP/1.1
...
Authorization: Bearer xxxxxxxxxxxxxxxxxx
...
{
   "parameters":{
      "requester": <requester_identifer>,
      "clientType":"aPlugin",
      "extraUri":"/trackers/<deviceId>/geolocations?
            order=asc&startTime=1662991200001&endTime=1663077599999&
            isSummary=true&limit=200",
      "method":"GET",
      "encodedHeaders": <base64_string>,
      "requesterToken": <base64_string>,
      "encodedBody":"",
      "clientVersion":"1",
      "uri":"/trackerapi"
   }
}
```

Figure 7: An example of a location history request

5.1 SmartTags

5.1.1 SmartTag Impersonation. We have created a proof-of-concept impersonation script for a SmartTag in python. This was created based on the reverse-engineered findings of SmartTag's firmware and SmartThings.apk to allow a device with the script running to operate like a legitimate registered SmartTag.

To successfully impersonate a SmartTag, so that it can be registered and participate in Samsung OF network, the impersonation would need to somehow successfully pass the finalization stage (see §4.3.5), so that a shared secret is established between the tag and the server. Since the tag stores private-public key pair of a legitimate SmartTag is embedded in the hardware and it is not explicitly exchanged in any stage of the registration process, with either the owner phone or the server, it cannot be obtained easily without performing a hardware-level attack on the tag. However, we can still obtain the shared secret by examining the data exchanges between the owner phone and the server, since this shared secret is sent explicitly by the server to the owner phone, sidestepping the need for extracting the private-public key of the tag.

Due to the need to establish a shared secret with the server, the full impersonation of a registered tag not belonging to the attacker is a minor concern. In our experiments, this was mainly useful for further runtime analyses of the interaction between the owner's phone, the helper devices and the (impersonated) tag. For example, we could modify the impersonated tag to always advertise in OFFLINE mode and observe the response of helper devices and to examine the behaviour of the anti-tracking algorithm implemented by Samsung.

5.1.2 Owner Device - Tag Interaction. We now examine the subprotocols that involve an owner device and a tag it owns. Assuming that

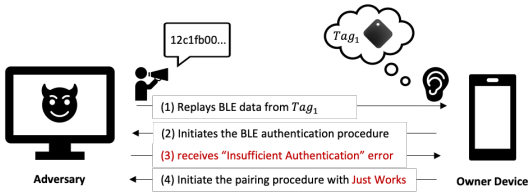

Figure 8: The silent pairing attack for Owner Devices

Unintended Silent Pairing with an Owner Device. In some circumstances, it is possible to impersonate a SmartTag to its owner device and pair silently with the owner device. This attack relies on the following pairing behaviour in the BLE specification: if a central device reads from or writes to a characteristic in a peripheral, and gets an 'Insufficient Authentication' error (error code 0x05), then the central will initiate a pairing procedure with the peripheral. In this scenario, the attacker is the peripheral, impersonating a SmartTag, and the central is the owner device. This behaviour was exploited in some prior work [20, 22, 23] to initiate an unintended pairing. As noted in [23], the attacker can influence the association method for the pairing, e.g., to force a downgrade of the pairing method to an insecure one (i.e., the Just Works association method). For the attack to work, the adversary needs to impersonate the GATT profile of a SmartTag and replay the latest BLE advertisement of the SmartTag to be impersonated to trick the owner device into initiating the BLE Authentication process with the tag. As described in §4.3.1, the BLE authentication starts with the owner device writing to the `NONCE_CHAR` of the SmartTag's GATT server. Therefore, by setting the write permission for the `NONCE_CHAR` of the impersonated GATT profile to `encrypted-write` would trigger the 'Insufficient Authentication' error upon write request to this characteristic. Figure 8 provides a summary of the attack flow.

In most versions of Android prior to November 2020 patch,[11] the pairing is performed silently whenever Just Works is used [20]. Some of the older models of Samsung devices, e.g., those running Android 8 or earlier versions, are vulnerable, since the Nov 2020 patch was not available for those devices. For example, we found that Samsung Galaxy 7, which has the September 2020 patch in its final firmware update, is vulnerable to this attack. But even with the patch, the attacker could still trick the owner to accept the pairing request via social engineering.

If this attack succeeds, the attacker would posess the IRK and the identity address of the owner's phone, which would allow the attacker to track the owner's phone. For example, if the owner's phone advertises a GATT profile, the IRK can be used to de-anonymize the BLE MAC address used in the advertisement. If the owner's phone does not advertise any GATT profiles, in most Android phones, we found that they will still respond to a 'ping' request at its identity address, if its Bluetooth adapter is turned on.

5.1.3 Helper Device - Tag Interaction. We now examine the interactions between a tag and another device that is not the owner device (e.g., a helper device or an adversary controlled device).

GATT server leaking sensitive data. We have found sensitive information leaked by two characteristics under the onboarding service (UUID FD59): the `IDENTIFIER` characteristic contains the identity MAC address/serial number of a tag, and the `HASHED_SERIAL_NUMBER` characteristic contains the SHA256 hash of the serial number. Both values are static and unique for each tag and readable by any connected device.

The `SUPPORTED_CIPHER` characteristic under the authentication service is readable and writable. It contains a default value: "AES_-128-CBC-PKCS7Padding", which specifies the cipher being used during the BLE authentication procedure. It has been observed that writing custom values to this characteristic would overwrite the default value that persists until the tag is restarted.

Therefore, an adversary in proximity to an advertising SmartTag can use either value as the identifier to de-anonymize the identity of the tag.

Hence, an adversary in proximity to a tag can overwrite this characteristic with a custom identifier (i.e., tag1, tag2) and use it to de-anonymize a tag.

DFU device reboot. The Galaxy SmartTag has a DFU Service (UUID FE59), which uses the Buttonless Secure DFU service module from Nordic Semiconductor. This service is intended to be used for secure over-the-air firmware updates. Any device can make a SmartTag enter the DFU mode by:

- enabling Buttonless DFU characteristic indication
- writing byte 0x01 to the Buttonless DFU characteristic

In DFU mode, the tag advertises on a Random Static MAC address with the device name "DFUTarg" and waits to receive new application firmware packages over-the-air. If no data is received over a short period, the tag will reboot into the application mode and resume its regular operation. It takes approximately two minutes for a DFU mode tag to reboot back into application mode if no firmware package was received.

[11] See https://source.android.com/security/bulletin/2020-11-01

Thus, an adversary in proximity to a registered SmartTag can abuse the DFU feature to interfere with the OF operation of the tag by repetitively forcing it to enter the DFU mode.

In addition, this attack allows an adversary to reveal the identity MAC address of a registered tag. In application mode, the identity MAC address $addr_1$ of a registered SmartTag is hidden by RPAs. When the tag operates in DFU mode, the static MAC address $addr_2$ it advertises on equals to $addr_1$ plus one, e.g., if $addr_2$ is observed to be 11:22:33:44:55:66, it can be inferred that $addr_1$ is 11:22:33:44:55:65.

Unintended bonding with a SmartTag. The update to firmware 1.02.06 introduces a new vulnerability that was not there in the previous version. The SmartTag with this new firmware appears to accept pairing request, using the Just Works association mode. This allows the attacker to obtain the IRK and the identity address of the tag silently without alerting the owner device.

The IRK can then be used to resolve the RPA the tag uses when advertising its payload. The IRK appears to be persistent across reboot and across account switching. So removing the tag from a Samsung account and pairing with another account does not reset the IRK. The possession of the IRK allows a more stealthy tracking of the tag, as the attacker does not need to connect to tag; they simply observe the RPA used to advertise the payload and de-anonymise it using the IRK. This can even be done offline, e.g., the attacker can collect the results of passive scans in a log file, and later de-anonymises the RPAs found in the log.

5.2 Shared Vulnerabilities

The next two vulnerabilities apply to both FMM devices and SmartTags.

5.2.1 Helper Device - Lost Device Interaction.

Fake Location Report by Helper Devices. The HTTPS communication involved in the location report process is secure. However, it was possible for a malicious owner of a Helper Device to gain the MitM position without breaking the HTTPS communication. This can be done by setting up a proxy server between the Helper Device and the remote server using tools such as BurpSuite.

If a lost tag is nearby, the adversary's Helper Device would pick up the BLE data broadcasted by the tag and prepare to issue a legitimate location report request to the `chaser` domain of Samsung.

With the Burp proxy server, the adversary can intercept this location report request and send it to the Burp repeater tool to modify certain parameters of the location report, allowing a fake report to be submitted to the server.

For instance, the adversary can customize the latitude and longitude values so that when the Owner Device receives the location update, their lost tag will appear in the wrong place on the map, as shown in Figure 9.

5.2.2 Online Helper Device - Server Communication.

Location Tracking by Service Operator. The OF protocol design for SmartTags allows Samsung, as the service operator, to track the locations of devices in the OF network using sensitive information stored on the server/presented during the server communication process.

Figure 9: Fake location being updated on the Owner Device

Samsung can access the reported locations of any SmartTag. The shared secret, privacy ID configuration, and other data provided by Samsung's server during the registration phase of a tag allow Samsung to associate each location report with the online profile of the tag by matching the private ID in the BLE data contained in the report with all the private IDs stored in its database. Then, the server can associate the online profile with its owner's Samsung account. This capability has been confirmed by our analysis of the location pulling process (see §4.5.3). The location request data sent by the owner device contains the bearer token and the deviceId that can be linked to the owner's account. After the Owner Device makes a location pulling request, the server responds with a list of GPS locations and timestamps, which indicate that the server is able to associate those location reports with the provided deviceId and the bearer token.

The location reporting process also leaves owners and helpers in the OF network vulnerable to potential location tracking from the operator. For a tag connected to its Owner Device, the Owner Device would act as a reporter that frequently reports the tag's location to the server. When a tag is lost, nearby Helper Devices would report its location to the server. To send a location report, the Helper Device must first obtain a JWE token, which is valid for 32 hours, and attach that token in its location reports. So all location reports within this 32-hour period can be all linked to the same device.

We note that the process for obtaining a JWE token involves answering a challenge from the server, by encrypting a nonce, chosen by the server, with a private signing key stored in the Helper Device. It is not clear how unique this private signing key is. Our very limited tests show that at least two distinct phones seem to share the same signing key, so this limits the possibility of using the signature as an identifying information.

6 DISCUSSIONS

We now discuss the impact of our findings in terms of addressing the research questions we posed in Section 1.

(RQ1) Identification of an FMM device or a SmartTag. Both FMM devices and SmartTags can be identified through their BLE behavior: An FMM device advertises identifiers chosen from a deterministically generated small privacy pool (size 51). Experiments have shown that an adversary can collect all 51 identifiers within a few days through BLE passive scanning and pattern match the identifier broadcasted by the device with collected identifiers to de-anonymize the device in the future. The SmartTag has a larger privacy ID pool, so the attack for FMM devices no longer applies. However, we have found that the SmartTag's GATT server leaks static and unique data, which can be used as the identifier to de-anonymize an advertising tag (see §5.1.3).

(RQ2) Unwanted tracking. There have not been any mature tracking prevention technologies to protect people from malicious trackers that leverage Samsung's OF network. Samsung SmartThings app has a feature that allows the user to scan for unknown SmartTags, and with the latest SmartThings app (version 1.7.89.25 at the time of writing) it supports background scanning as well. We have not yet analysed this feature completely. Our limited preliminary tests showed that it did trigger alerts of unknown tags. However, it does not seem to support detection of offline devices other than SmartTags, so potentially, an adversary could create a custom tracking device that simulates lost phones rather than lost tags. We have not yet tested this possibility. We also note that this unknown tag scanning feature is only accessible to users with the SmartThings application and a Samsung account, leaving users of phones from other vendors at risk. However, our analysis also uncovered several ways in which a tag can be identified, relatively easily in comparison to AirTags, as addressed in the RQ1 above. We have implemented a proof-of-concept tool to identify unwanted tracking for different types of Samsung OF devices based on our findings.

We note that the answers to (RQ1) and (RQ2) may appear contradictory on the surface, as measures to prevent the identification of an OF device may work against the measures against unwanted tracking prevention. In the case of Samsung OF, because the Samsung cloud server can link the advertisements broadcast by a malicious tag to its owner, Samsung is in a position to identify definitively the case of an unwanted tracking, and in principle can disable the tracking support for the malicious tag (although we currently do not know whether such a feature exists on the server side) and attribute the unwanted tracking offense to a particular Samsung account. This is not possible with Apple Find My network and AirTags, as due to the end-to-end privacy feature of Apple Find My protocol, Apple OF network cannot link the BLE advertisement of a malicious tag to its owner to allow for a targeted prevention. Although it is possible to identify the owner of an AirTag through its serial number, this would require physical access to the tag and it assumes that it is a genuine tag, so not a custom tag (that can be created through, e.g., the use of the OpenHaystack framework).

However, from [8, 14, 19], we know that with lost AirTags, the advertisement data and the MAC address of the tag rotate infrequently (around 24 hours or so), which helps in determining whether a (lost) tag is indeed tracking a particular user through its MAC address or advertisement data. In the case of lost FMM devices, the advertisement data and the MAC address of the tags rotate in sync and relatively frequently (around every 15 minutes), so it is generally difficult to determine unwanted tracking based on advertisement

data or MAC address alone, especially in a crowded environment with multiple lost devices around. With SmartTags, the situation is a bit mixed. For SmartTags in an overmature state, the privacy ID rotation is less frequent and seems to match that of AirTags, so it will be relatively easy to distinguish such a tag from other tags around. However, as with AirTags, an attacker could create a custom tag that rotates its privacy ID frequently enough to avoid its advertisement data being linked.

(RQ3) *End-to-end location privacy.* The OF protocol design allows the service operator (Samsung) to access the reported locations of any OF devices (FMM devices and tags) and associate the locations with the device owner through the sensitive data stored on the server, e.g., encryption keys and privacy configurations. Moreover, helper devices are also vulnerable to tracking, as each location report contains a JWE authentication token issued by the location server that is unique for each helper device and can be linked for an extended period (see §4.2.3).

(RQ4) *Location report integrity.* Our tests indicate that both the helper devices and the location server do not check whether a location report accurately reflects the actual location of a lost device. The helper device, when scanning for lost devices, does not attempt any validation on the scanned device whether it is a genuine Samsung phone or a genuine SmartTag; as long as the BLE advertisement data is in the expected format, it will be forwarded to the location server. The location server validates that the BLE advertisement data is genuine, i.e., that it belongs to a device of a registered user, but it assumes that the helper device has reported the location truthfully. From our tests, it seems that the location server does not attempt to perform any further validation; for example, it is possible to create two location reports for the same tag that show the tag is detected within seconds of each other at two locations that are geographically so distant that it is practically impossible to reach one location from another within the time frame it was detected. This can be achieved, e.g., by relaying BLE data from a tag between colluding adversaries in different locations.

7 CONCLUSION

In this work, the OF and device management protocols for FMM devices and SmartTags have been thoroughly analyzed, and a security and privacy analysis was performed. Our analysis of the protocols' design and implementation has identified several flaws, allowing each of the research questions to be answered definitively.

We have also discovered vulnerabilities outside the scope defined by the proposed research questions, including multiple other flaws related to the GATT server implementation for SmartTags, and the flaw in the registration protocol that allows an attacker to register a SmartTag of someone else without knowing its ECDH private key.

SmartTags with firmware versions 1.01.26 and 1.02.06 and FMM devices with FMM versions below 7.2.24.12 were used in our analysis, while changes introduced by newer releases of the firmware/FMM software have not been investigated in detail here. Therefore, some of our findings and analysis results may not apply to devices/tags with higher versions. However, at the time of writing, our tests show that devices or tags with older firmware/software versions can still participate in the OF network. Existing users of SmartTags and other FMM devices who have the option to upgrade the firmware/apps on their devices are encouraged to do so to mitigate some of the issues we discuss here.

Among the issues we discussed, of great concern is the possibility of using SmartTags and similar trackers, such as AirTags and Tile trackers, or even custom tracking devices leveraging on these offline finding networks for unwanted tracking. The current fragmented approach to anti-stalking features leaves a significant number of people vulnerable to unwanted tracking without an effective mean for detecting it. For future work, we plan to investigate ways to detect unwanted tracking that are effective against a variety of OF networks, leveraging on existing efforts such as AirGuard [14].

REFERENCES

[1] Apple. 2022. Find My and Privacy. https://www.apple.com/au/legal/privacy/data/en/find-my/ Accessed 01-Oct-2022.

[2] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3958)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer, 207–228. https://doi.org/10.1007/11745853_14

[3] Luca Bongiorni. 2021. Samsung SmartTag Hack. https://github.com/whid-injector/Samsung-SmartTag-Hack.

[4] Guillaume Celosia and Mathieu Cunche. 2019. Fingerprinting Bluetooth-Low-Energy Devices Based on the Generic Attribute Profile. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019*, Peng Liu and Yuqing Zhang (Eds.). ACM, 24–31. https://doi.org/10.1145/3338507.3358617

[5] Guillaume Celosia and Mathieu Cunche. 2019. Saving private addresses: an analysis of privacy issues in the bluetooth-low-energy advertising mechanism. In *MobiQuitous 2019, Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, Texas, USA, November 12-14, 2019*, H. Vincent Poor, Zhu Han, Dario Pompili, Zhi Sun, and Miao Pan (Eds.). ACM, 444–453. https://doi.org/10.1145/3360774.3360777

[6] Guillaume Celosia and Mathieu Cunche. 2020. Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols. *Proc. Priv. Enhancing Technol.* 2020, 1 (2020), 26–46. https://doi.org/10.2478/popets-2020-0003

[7] Char49. 2019. Samsung Find My Mobile vulnerability. https://char49.com/tech-reports/fmmx1-report.pdf. Accessed 01-Oct-2022.

[8] James Clayton and Jasmin Dyer. 2022. Apple AirTags - A perfect tool for stalking. https://www.bbc.com/news/technology-60004257. Accessed 01-Oct-2022.

[9] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer. https://doi.org/10.1007/978-3-662-04722-4

[10] Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Trans. Inf. Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[11] Core Specification Working Group. 2021. Bluetooth Core Specification. (2021). https://www.bluetooth.com/specifications/specs/core-specification-5-3/

[12] Samsung Group. 2022. Samsung SmartThings Find Hits New Milestone With 200 Million Nodes Helping Find Lost Devices. https://news.samsung.com/global/samsung-smartthings-find-hits-new-milestone-with-200-million-nodes-helping-find-lost-devices. Accessed: 2022-07-28.

[13] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. 2010. *Guide to Elliptic Curve Cryptography* (1st ed.). Springer Publishing Company, Incorporated.

[14] Alexander Heinrich, Niklas Bittner, and Matthias Hollick. 2022. AirGuard - Protecting Android Users From Stalking Attacks By Apple Find My Devices. *CoRR* abs/2202.11813 (2022). arXiv:2202.11813 https://arxiv.org/abs/2202.11813

[15] Alexander Heinrich, Milan Stute, Tim Kornhuber, and Matthias Hollick. 2021. Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System. *Proc. Priv. Enhancing Technol.* 2021, 3 (2021), 227–245. https://doi.org/10.2478/popets-2021-0045

[16] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik C. Rye, Brandon Sipes, and Sam Teplov. 2019. Handoff All Your Privacy - A Review of Apple's Bluetooth Low Energy Continuity Protocol. *Proc. Priv. Enhancing Technol.* 2019, 4 (2019), 34–53. https://doi.org/10.2478/popets-2019-0057

[17] Travis Mayberry, Ellis Fenske, Dane Brown, Jeremy Martin, Christine Fossaceca, Erik C. Rye, Sam Teplov, and Lucas Foppe. 2021. Who Tracks the Trackers?:

Circumventing Apple's Anti-Tracking Alerts in the Find My Network. In *WPES '21: Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society, Virtual Event, Korea, 15 November 2021*. ACM, 181–186. https://doi.org/10.1145/3463676.3485616

[18] RFC5652. 2009. Cryptographic Message Syntax (CMS). https://www.rfc-editor.org/rfc/rfc5652. Accessed 01-Oct-2022.

[19] Thomas Roth, Fabian Freyer, Matthias Hollick, and Jiska Classen. 2022. AirTag of the Clones: Shenanigans with Liberated Item Finders. In *15th USENIX Workshop on Offensive Technologies, WOOT 2022*.

[20] Alwen Tiu and Jim Mussared. 2020. A silent pairing issue in Bluetooth-based contact tracing apps. https://github.com/alwentiu/COVIDSafe-CVE-2020-12856/blob/master/CVE-2020-12856-19-June-2020.pdf.

[21] Mira Weller, Jiska Classen, Fabian Ullrich, Denis Waßmann, and Erik Tews. 2020. Lost and found: stopping bluetooth finders from leaking private information. In *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Linz, Austria, July 8-10, 2020*, René Mayrhofer and Michael Roland (Eds.). ACM, 184–194. https://doi.org/10.1145/3395351.3399422

[22] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. 2019. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/badbluetooth-breaking-android-security-mechanisms-via-malicious-bluetooth-peripherals/

[23] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2020. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 37–54.